
南開大學
Nankai University

计算机学院

并行程序设计实验报告

# 实验五：mpi 编程（口令猜测）

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 6 月 8 日

# 目录

# 1 问题重述

PCFG（Probabilistic Context-Free Grammar，概率上下文无关文法）口令猜测是一种基于语言建模原理的密码猜测攻击技术，它结合了统计学习和文法规则，用于高效生成可能的密码候选列表，优于传统暴力破解或字典攻击。其核心思想是：利用密码样本学习密码结构和模式的概率分布，构建一个概率文法模型，再据此生成猜测结果，按概率从高到低进行尝试。

在理解了实验指导书上的内容之后，我们可以归纳出其基本需要实现以下几个过程：

1. 分析训练集：首先会对训练集进行分析，提取一些密码的结构并计算概率，比如 L8D3S3 这种，代表这个密码由 8 个字母 +3 个数字 +3 个符号组成，并且会计算出其在训练集中出现的概率。

2. 建立概率模型：为每种模式生成一个文法规则，并为每种模式及其组成部分（字母、数字、符号）计算出现概率。也就是计算某个密码出现的概率，分为上一过程提到的模式概率和组件概率（也就是向其中填入真正的字符信息）。比如说 L8D1S1 的概率是 0.2，在 L8 中，12345678 出现的概率为 0.5，在 D1 中 a 出现的概率为 0.1，在 S1 中！出现的概率为 0.1，那么密码 12345678a！的总概率就为 0.5*0.2*0.1*0.1（条件概率）。

3. 生成口令候选：根据概率文法生成新的口令，优先生成高概率组合。

4. 猜测过程：生成的口令候选从高概率到低概率依次尝试，大幅减少猜测空间，提高成功率。

另外，PT 指的是一种拆分结构（如 L6S1），其中的每个部分叫 Segment。

原理部分还是如上所示，在本次实验中，我们需要**基于 mpi 实现口令猜测算法的并行化**。

# 2 并行算法设计

## 2.1 并行算法思路

思考这个问题，我们需要先思考，为什么这两个 for 可以进行并行化？根本原因是因为这些数据是并行的，没有相互依赖，并且这个过程是一个读多写少的过程，不涉及全局状态改变，可以通过加锁操作，不会导致数据冲突。

另外，我们为什么要进行并行化优化呢？原因是单线程处理时，每次扩展和计算概率的复杂度较高，且需要维护和排序大量节点。将扩展任务分配到多个进程中，可以极大缩短整体搜索时间。

由于每个节点的扩展与概率计算是独立操作，且扩展后生成的新节点只需要归并回主队列即可，适合用 MPI 进行分布式并行计算。

我们并行化的思路，就是分配多个进程，把这个任务量比较大的任务细化，分配给多个进程，每个进程处理一部分数据，并在最后总和各个进程完成的任务。

该算法采用主从架构实现 MPI 并行化。具体流程如下：

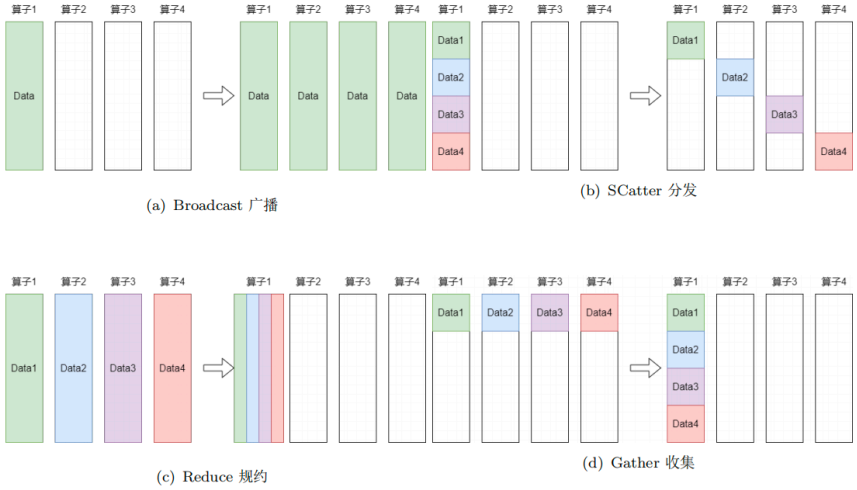


图 2.1: MPI 并行化流程

1. 初始化 MPI 环境，获取当前进程编号（rank）和总进程数。
2. 主进程（rank 0）维护全局任务队列，存储待扩展的密码节点。
3. 主进程将任务队列按批次拆分成多个子任务，通过 MPI 发送给各个工作进程。
4. 工作进程接收任务后，独立进行节点扩展和概率计算，并通过 MPI 将结果发送回主进程。
5. 主进程接收各工作进程返回的结果，合并回全局队列，更新搜索状态。
6. 重复分发任务与结果合并的过程，直至满足终止条件。
7. 所有进程完成后，释放 MPI 资源，结束计算。

该并行流程有效利用多核计算资源，通过任务分割和结果汇总，实现密码猜测的加速。

## 2.2 并行算法具体实现

Generate_mpi 这个函数是整个 MPI 并行计算的核心。它的任务就是根据当前的 PT 节点，生成所有对应的猜测组合，并且把任务合理分配给各个 MPI 进程来并行计算。

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

void PriorityQueue::Generate_mpi(PT pt)
{
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    CalProb(pt);
```

首先，函数会通过 MPI 的接口拿到当前进程的编号 rank 和总进程数 size，这是分配任务的基础。然后，它调用 CalProb（在上次实验已经详细分析过，这里不再赘述）来算出当前 PT 的概率，这个概率会用来计算最终每个猜测组合的概率。

```
    if (pt.content.size() == 1) {
        segment *a;
        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])];
        }

        int total = pt.max_indices[0];

        // 改进的负载均衡: 动态分配
        int base_chunk = total / size;
        int remainder = total % size;
        int start_idx, end_idx;

        if (rank < remainder) {
            // 前remainder个进程多分配一个任务
            start_idx = rank * (base_chunk + 1);
            end_idx = start_idx + base_chunk + 1;
        } else {
            // 后面的进程分配base_chunk个任务
            start_idx = remainder * (base_chunk + 1) + (rank - remainder) *
                base_chunk;
            end_idx = start_idx + base_chunk;
        }
```

接着，函数判断这个 PT 的内容段数。如果只有一段内容，也就是只需要猜测一个单独元素，情况比较简单。它先算出这段内容的所有可能取值的总数，比如说这个位置能出现多少不同的字母或数字。然后，它把这个总数平均分成 size 份，分配给每个进程一个区间，让每个进程只负责计算自己区间内的所有猜测。这样不同的进程就不会重复计算，从而就保证了负载均衡。

```
        // 生成密码
        for (int i = start_idx; i < end_idx; i++) {
            guesses.push_back(a->ordered_values[i]);
        }
        // 本地计数
        int local_count = (end_idx - start_idx);

        // MPI合并: 收集所有进程的计数
        int global_total = 0;
        MPI_Allreduce(&local_count, &global_total, 1, MPI_INT, MPI_SUM,
```

```
            MPI_COMM_WORLD);

        // 更新为全局总计数（类似原始Generate函数的total_guesses += total）
        total_guesses = global_total;
    }
```

然后，每个进程会遍历自己负责的索引区间，把每个索引转换成实际猜测的内容，计算概率，并把结果保存起来。这部分工作各进程间完全独立，所以不需要通信。在所有进程完成各自计算后，我们可以使用 MPI 的 Allreduce 操作把所有进程产生的猜测数量汇总起来，保证每个进程都知道整个系统中猜测的总数。这一步保证了进程间同步，也方便后续步骤管理猜测等。

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

如果 PT 有多段内容的话，情况就相对稍微复杂一点。函数先把除了最后一段之外的内容段做一个前缀组合，也就是说先去确定前面所有 segment 的猜测值，然后只对最后一个 segment 的内容做任务划分。这样就可以把多段组合的任务拆成“前缀 + 后缀”的结构，每个进程只负责不同后缀的猜测部分。

```
        int total = pt.max_indices[pt.content.size() - 1];
```

```
2
3           // 改进的负载均衡: 动态分配
4           int base_chunk = total / size;
5           int remainder = total % size;
6           int start_idx, end_idx;
7
8           if (rank < remainder) {
9               // 前remainder个进程多分配一个任务
10              start_idx = rank * (base_chunk + 1);
11              end_idx = start_idx + base_chunk + 1;
12          } else {
13              // 后面的进程分配base_chunk个任务
14              start_idx = remainder * (base_chunk + 1) + (rank - remainder) *
                    base_chunk;
15              end_idx = start_idx + base_chunk;
16          }
17
18          // 生成密码
19          for (int i = start_idx; i < end_idx; i++) {
20              guesses.push_back(prefix + a->ordered_values[i]);
21          }
```

后续同样把最后一段的猜测空间均分，分配给不同进程，让每个进程在固定的前缀基础上遍历自己负责的后缀范围，从而生成完整猜测，计算概率。

```
1
2           // 本地计数
3           int local_count = (end_idx - start_idx);
4
5           // MPI合并: 收集所有进程的计数
6           int global_total = 0;
7           MPI_Allreduce(&local_count, &global_total, 1, MPI_INT, MPI_SUM,
                MPI_COMM_WORLD);
8
9           // 更新为全局总计数 (类似原始Generate函数的total_guesses += total)
10          total_guesses = global_total;
11      }
12  }
```

在所有进程完成各自计算后，再次调用 MPI 的 Allreduce 操作把所有进程产生的猜测数量汇总起来，保证每个进程都知道整个系统中猜测的总数。这一步保证了进程间同步，也方便后续步骤管理猜测。

总结来说，我们的 Generate_mpi 函数通过合理划分计算任务，把复杂的猜测组合生成工作均匀分配给所有 MPI 进程，既避免重复计算，又能充分利用集群资源。同时，它利用 MPI 通信来同步和汇总计算结果，确保并行计算的正确和高效。

除此以外，我们还要修改 main.cpp 中的 main 函数，以适配 mpi 并行化算法的框架，其中一个重点就是要进行进程的初始化 MPI_Init(&argc, &argv) 和销毁 MPI_Finalize() 等。

可以看到，我们给 main 函数设置了两个参数，使得我们可以通过命令行向程序中传入 mpi 初始化的参数，以帮助 mpi 解析命令行参数。

```cpp
#include "PCFG.h"
#include <chrono>
#include <fstream>
#include "md5.h"
#include <iomanip>
#include <mpi.h>
using namespace std;
using namespace chrono;

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);

    // 使用MPI计时工具
    double mpi_time_start_total = MPI_Wtime();  // 总体开始时间
    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;

    // MPI计时：训练阶段
    double mpi_time_train_start = MPI_Wtime();
    if (rank == 0) {
        cout << "Starting model training..." << endl;
    }

    q.m.train("input/Rockyou-singleLined-full.txt");
    q.m.order();

    double mpi_time_train_end = MPI_Wtime();
    time_train = mpi_time_train_end - mpi_time_train_start;

    if (rank == 0) {
        cout << "Model training completed in " << time_train << " seconds" << endl;
    }

    // 等待所有进程完成训练
    MPI_Barrier(MPI_COMM_WORLD);

    q.init();

    if (rank == 0) {
        cout << "Starting password generation with " << size << " MPI processes..."
```

```
            << endl;
49      }
50
51      int curr_num = 0;
52
53      // MPI计时：猜测生成阶段
54      double mpi_time_guess_start = MPI_Wtime();
55
56      // 由于需要定期清空内存，我们在这里记录已生成的猜测总数
57      int history = 0;
58
59      while (!q.priority.empty())
60      {
61          q.PopNext();  // 已经使用Generate_mpi进行MPI并行化
62
63          // 收集所有进程的猜测总数
64          int local_guesses = q.guesses.size();
65          int global_guesses = 0;
66          MPI_Allreduce(&local_guesses, &global_guesses, 1, MPI_INT, MPI_SUM,
                MPI_COMM_WORLD);
67          q.total_guesses = global_guesses;
68
69          if (q.total_guesses - curr_num >= 1000000)
70          {
71              if (rank == 0) {
72                  cout << "Process " << rank << " - Global guesses generated: " <<
                        history + q.total_guesses << endl;
73              }
74              curr_num = q.total_guesses;
75
76              // 在此处更改实验生成的猜测上限
77              int generate_n = 30000000;
78              if (history + q.total_guesses > generate_n)
79              {
80                  double mpi_time_guess_end = MPI_Wtime();
81                  time_guess = mpi_time_guess_end - mpi_time_guess_start;
82
83                  if (rank == 0) {
84                      cout << "=== MPI Timing Results (Process " << rank << ") ===" <<
                            endl;
85                      cout << "Guess time: " << time_guess - time_hash << " seconds" <<
                            endl;
86                      cout << "Hash time: " << time_hash << " seconds" << endl;
87                      cout << "Train time: " << time_train << " seconds" << endl;
88
89                      double total_time = MPI_Wtime() - mpi_time_start_total;
90                      cout << "Total execution time: " << total_time << " seconds" <<
                            endl;
91                  }
```

```
92                  break;
93              }
94          }
95  //为了避免内存超限，我们在q.guesses中口令达到一定数目时，将其中的所有口令取出并且进行哈希
96          if (curr_num > 1000000)
97          {
98              // MPI计时：哈希阶段
99              double mpi_time_hash_start = MPI_Wtime();
100
101             // 预先计算批次数量，减少vector的动态调整
102             const int batchSize = 4;
103             const int numFullBatches = q.guesses.size() / batchSize;
104             const int remainder = q.guesses.size() % batchSize;
105
106             // 分配状态数组
107             bit32 *state[batchSize];
108             for (int i = 0; i < batchSize; ++i) {
109                 state[i] = new bit32[4];
110             }
111             string inputs[batchSize];
112
113             // 处理完整批次
114             for (int batch = 0; batch < numFullBatches; ++batch) {
115                 for (int i = 0; i < batchSize; ++i) {
116                     inputs[i] = q.guesses[batch * batchSize + i];
117                 }
118                 MD5Hash_NEON(inputs, state);
119             }
120
121             // 处理剩余项
122             if (remainder > 0) {
123                 string inputs[batchSize];
124                 for (int i = 0; i < remainder; ++i) {
125                     inputs[i] = q.guesses[numFullBatches * batchSize + i];
126                 }
127                 for (int i = 0; i < remainder; ++i) {
128                     bit32 singleState[4];
129                     MD5Hash(inputs[i], singleState);
130                 }
131             }
132
133             // 释放内存
134             for (int i = 0; i < batchSize; ++i) {
135                 delete[] state[i];
136             }
137
138             // MPI计时：哈希结束
139             double mpi_time_hash_end = MPI_Wtime();
140             double hash_duration = mpi_time_hash_end - mpi_time_hash_start;
```

```
141             time_hash += hash_duration;
142
143             if (rank == 0 && curr_num % 5000000 == 0) {
144                 cout << "Process " << rank << " - Hash batch completed in " <<
                        hash_duration << " seconds" << endl;
145             }
146
147             // 记录已经生成的口令总数
148             history += curr_num;
149             curr_num = 0;
150             q.guesses.clear();
151         }
152     }
153
154     // 最终的MPI计时统计
155     double mpi_time_end_total = MPI_Wtime();
156     double total_execution_time = mpi_time_end_total - mpi_time_start_total;
157
158     // 收集所有进程的计时信息
159     double max_total_time, min_total_time, avg_total_time;
160     MPI_Reduce(&total_execution_time, &max_total_time, 1, MPI_DOUBLE, MPI_MAX, 0,
            MPI_COMM_WORLD);
161     MPI_Reduce(&total_execution_time, &min_total_time, 1, MPI_DOUBLE, MPI_MIN, 0,
            MPI_COMM_WORLD);
162     MPI_Reduce(&total_execution_time, &avg_total_time, 1, MPI_DOUBLE, MPI_SUM, 0,
            MPI_COMM_WORLD);
163
164     if (rank == 0) {
165         avg_total_time /= size;
166         cout << "\n=== Final MPI Timing Summary ===" << endl;
167         cout << "Number of MPI processes: " << size << endl;
168         cout << "Max execution time across all processes: " << max_total_time << "
                seconds" << endl;
169         cout << "Min execution time across all processes: " << min_total_time << "
                seconds" << endl;
170         cout << "Average execution time across all processes: " << avg_total_time <<
                " seconds" << endl;
171         cout << "Load balance efficiency: " << (min_total_time / max_total_time) *
                100 << "%" << endl;
172     }
173     MPI_Finalize();
174     return 0;
175 }
```

这段代码是我们整个 PCFG 密码猜测程序的主函数部分，这部分主要负责程序的初始化、模型训练、密码生成与哈希处理，并通过 MPI 实现并行计算。程序首先通过 MPI_Init 初始化 MPI 环境，确定当前进程编号（rank）和总进程数（size）。接下来，rank 为 0 的主进程会启动模型训练，并记录训练所耗时间。

训练完成后，所有进程进入密码生成阶段。程序会不断从优先队列中弹出新的猜测，并定期进行全局同步，统计所有进程生成的猜测总数。当生成数量达到一定的阈值时，会触发 MD5 哈希操作，对当前猜测集合执行 MD5 哈希。哈希过程也采用批处理方式，提高效率并避免内存溢出（我们第一个 simd 实验）。

程序设定了猜测上限（如三千万条），一旦达到上限，主进程 rank0 将会输出各阶段的耗时信息，包括训练时间、猜测时间、哈希时间和总运行时间。最后，还会收集所有进程的总耗时，统计最大、最小和平均执行时间，并计算负载均衡效率，从而评估并行效果和资源利用情况。

并且我们按照了报告里的要求，计时测试工具使用 MPI 进行计时（MPI_Wtime()），以获得更精确的时间测量。

### 2.3 并行算法的正确性验证

可以看到我们实现的并行算法的猜测正确数与串行算法完全相同，验证了并行算法逻辑上的正确性。

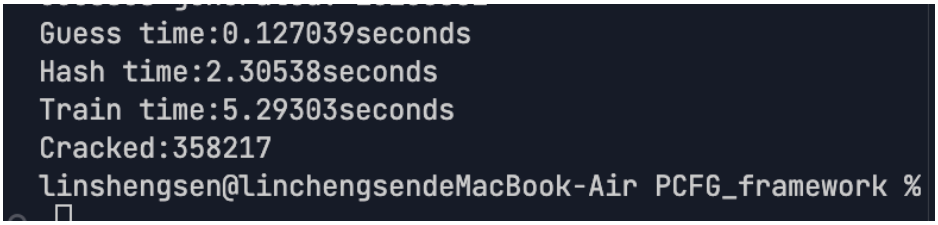


图 2.2: 串行

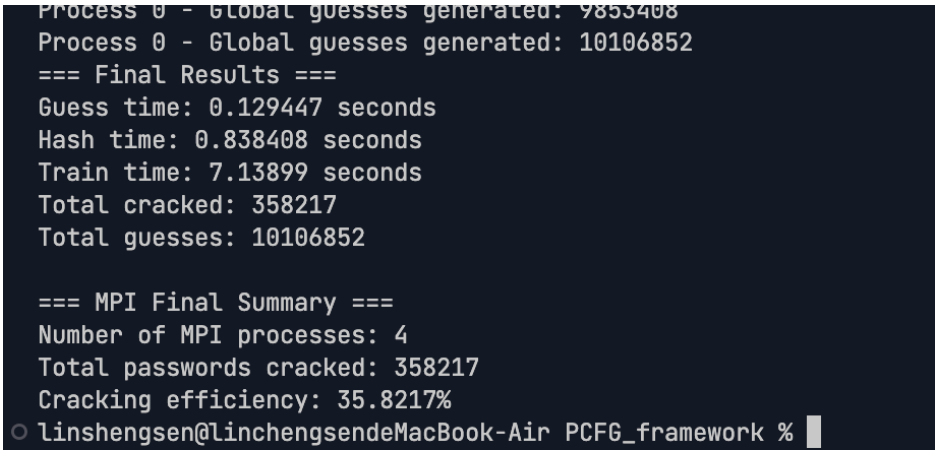


图 2.3: mpi 并行算法的正确性验证

## 3 性能对比

由于服务器不稳定 guess_time 一直在大幅度跳，所以这里在我自己的 mac 电脑上跑。

对于普通的串行程序，采用编译命令

```
// g++ main.cpp train.cpp guessing.cpp md5.cpp -o main
// g++ main.cpp train.cpp guessing.cpp md5.cpp -o main -O1
// g++ main.cpp train.cpp guessing.cpp md5.cpp -o main -O2
```

对于 mpi 多进程程序，采用编译命令为

```
//mpic++ main.cpp train.cpp guessing_mpi.cpp md5.cpp -o test_mpi_optimized -std=c++11
//mpic++ main.cpp train.cpp guessing_mpi.cpp md5.cpp -o test_mpi_optimized -O1
    -std=c++11
//mpic++ main.cpp train.cpp guessing_mpi.cpp md5.cpp -o test_mpi_optimized -O2
    -std=c++11
```

## 3.1 不开启编译优化的情况下

表 1: 串行与 mpi

| 问题规模 | 串行 hash | 并行 hash | 串行 guess | 并行 guess | hash 加速 | guess 加速 |
|---|---|---|---|---|---|---|
| 500w | 2.734s | 1.27569s | 0.852s | 0.857176s | 2.143 | 0.994 |
| 1000w | 6.207s | 2.85848s | 1.470s | 1.25911s | 2.171 | 1.167 |
| 2000w | 12.700s | 5.87321s | 2.986s | 2.71154s | 2.162 | 1.101 |
| 3000w | 19.027s | 8.80816s | 4.321s | 3.69126s | 2.160 | 1.171 |

在不开启编译优化的情况下，可以看到并行算法的 hash time 的时间已经远小于串行算法，加速比在 2.15 左右，这在一定程度上与我们之前的 hash time 加速比的结果不同（之前不开编译优化的情况下都是负优化，并行的 hash time 用时较串行来说更长），这里不同的原因应该主要就是我们的编译指令是 mpic++ 而不是之前的 g++，它虽然底层也是调用 g++，但默认带了一些额外的优化选项，或者链接了 MPI 库带来的性能提升，对循环展开、函数内联展开的效果比 g++ 编译器好得多，因此生成的代码效率更高。相比之下，之前直接用 g++ 编译时没有这些优化，导致并行代码开销大于收益。另外，guess time 也略优于串行算法，并且随着问题规模的增大，guess time 加速比不断增大，这与我们的自然认知一致，随着问题规模的增大，多个进程可以更充分地利用多核资源，整体计算速度也就越快。

## 3.2 O1 优化情况下

表 2: 串行与 mpi

| 问题规模 | 串行 hash | 并行 hash | 串行 guess | 并行 guess | hash 加速 | guess 加速 |
|---|---|---|---|---|---|---|
| 500w | 0.624s | 0.120479s | 0.090s | 0.07717s | 5.179 | 1.166 |
| 1000w | 1.396s | 0.267561s | 0.149s | 0.119078s | 5.218 | 1.251 |
| 2000w | 2.847s | 0.547634s | 0.295s | 0.230932s | 5.199 | 1.277 |
| 3000w | 4.308s | 0.827955s | 0.437s | 0.344645s | 5.203 | 1.268 |

在 O1 编译条件下，可以看到由于循环进一步进行了展开，hash time 实现了更明显的加速，平均加速比已经来到了 5.2，并且 guess time 也实现了加速，随着问题规模的增大加速效果越来越好，加速效果也比较显著。

### 3.3 O2 优化情况下

表 3: 串行与 mpi

| 问题规模 | 串行 hash | 并行 hash | 串行 guess | 并行 guess | hash 加速 | guess 加速 |
|---|---|---|---|---|---|---|
| 500w | 0.423s | 0.114356s | 0.088s | 0.082256s | 3.699 | 1.07 |
| 1000w | 0.945s | 0.289768s | 0.143s | 0.127741s | 3.261 | 1.119 |
| 2000w | 1.908s | 0.516342s | 0.286s | 0.217392s | 3.695 | 1.316 |
| 3000w | 2.871s | 0.802938s | 0.409s | 0.347614s | 3.576 | 1.177 |

在 O2 编译情况下，跟之前的 simd 编程的 hash time 一样，加速比都不如 O1 优化，对此的解释就是，应该是由于 O2 编译对于串行算法的加速超越了对于 simd 并行算法的加速，所以使得加速比在一定程度上进行了减小，但加速效果还是很显著的。guess time 还是依旧实现了加速，并随着问题规模增大加速效果越来越好。

基于以上的测试与观察，我发现随着问题规模的增大，多进程编程可以并行充分使用多个核资源，加速计算。

## 4 进阶要求的实现

### 4.1 实现相对串行算法的加速

这个结果是在服务器比较稳定的情况下截取的，可以看到，我们的算法都实现了相对串行算法的加速，无论是 hash time 还是 guess time 都优化了。(O2 编译 1000w 数据)

```
mpic++ main.cpp train.cpp guessing_mpi.cpp md5.cpp -o main -O2
```

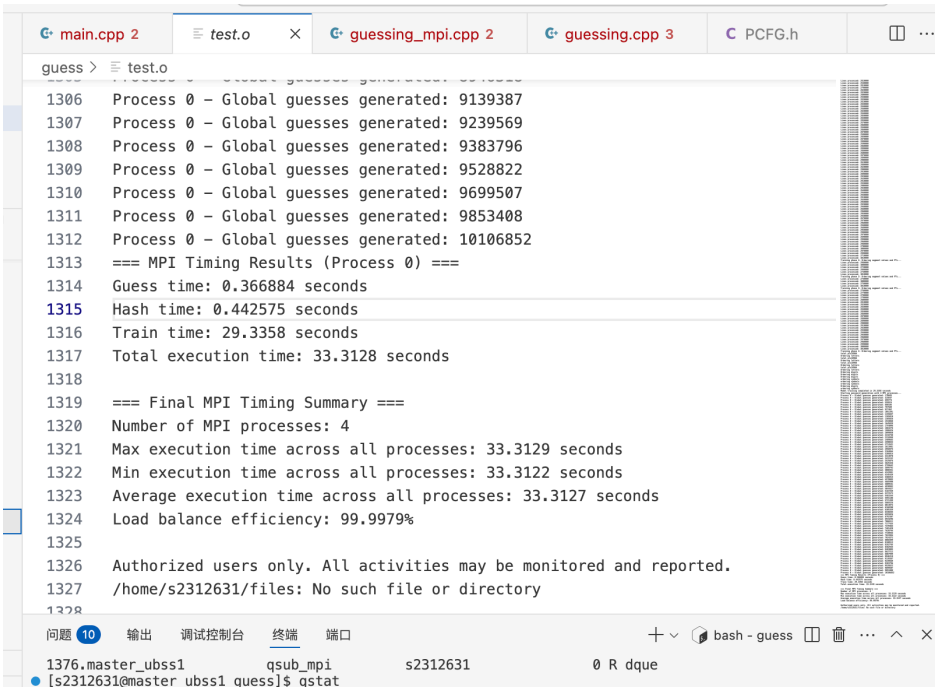


图 4.4: 并行 mpi 服务器测试

```
374  Guesses generated: 8282785
375  Guesses generated: 8382935
376  Guesses generated: 8483085
377  Guesses generated: 8627212
378  Guesses generated: 8807999
379  Guesses generated: 8946318
380  Guesses generated: 9139387
381  Guesses generated: 9239569
382  Guesses generated: 9383796
383  Guesses generated: 9528822
384  Guesses generated: 9699507
385  Guesses generated: 9853408
386  Guesses generated: 10106852
387  Guess time:0.590657seconds
388  Hash time:3.01137seconds
389  Train time:27.3431seconds
390
391  Authorized users only. All activities may be monitored and reported.
392  /home/s2312631/files: No such file or directory
393
```


图 4.5: 串行服务器测试

可以看到，串行算法的 hash time 为 3.01137s，guess time 为 0.590657s，并行算法的 hash time 为 0.442575s，guess 加速比为 6.804，guess 加速比 1.61。

## 4.2 一次性从优先队列中取出多个 PT

代码太长了就不粘贴了，代码放在最后的 github 链接中。

### 4.2.1 算法分析

在这一部分代码中，我们主要实现了对概率模板（PT）的批次处理，并且在处理过程中引入了 MPI 来实现并行加速。总体上来说，它的核心目标就是从优先队列里一次取出一批 PT，然后在多个进程之间分配任务，对这些 PT 进行扩展和处理，最后把新生成的 PT 再插回队列中，通过这个过程，我们可以高效地生成密码候选串。

```
function PopNextBatch_mpi(batch_size):
    获取MPI进程信息(rank, size)
    确定实际批量大小 = min(batch_size, 队列大小)

    从优先队列取出batch_size个PT
```

```
    并行处理这批PT

    收集所有新生成的PT
    计算新PT的概率
    移除已处理的PT
    将新PT按概率重新插入优先队列
end
```

```
function ProcessPTBatch_mpi(pt_batch):
    获取MPI进程信息(rank, size)

    计算每个进程分配的PT数量
    根据进程rank确定处理范围[start_pt, end_pt)

    for 分配给当前进程的每个PT:
        生成该PT的所有口令
        添加到本地猜测列表

    将本地猜测合并到全局猜测
    使用MPI_Allreduce同步总猜测数量
end
```

在我们新加的主函数 PopNextBatch_mpi 中，程序先根据实际情况确定本轮需要处理多少个 PT（注意不能超过队列里已有的数量），然后把这些 PT 按顺序拿出来放到一个数组里。拿出来之后，它就调用我们新定义的另外一个函数 ProcessPTBatch_mpi，这个函数的任务就是把这些 PT 平均分给多个进程去处理。每个进程只负责自己那一部分区间，从而把计算量分摊出去，提高整体效率。

```
function GenerateSinglePT_mpi(pt, pt_guesses):
    计算PT概率

    if PT只有一个段:
        根据段类型找到对应的segment
        生成该段的所有可能值
    else:
        构建前缀(除最后一段外的所有段)
        找到最后一段对应的segment
        为每个可能的最后段值生成完整口令

    将生成的口令添加到pt_guesses
end
```

在每个进程内部，实际的猜测生成工作是由 GenerateSinglePT_mpi 函数完成的。它根据每个 PT 的结构来判断要怎么生成密码串。比如说如果一个 PT 只有一个片段，它就直接根据这个片段的类型（是字母、数字还是符号）去找对应的备选字符串，一次性生成所有组合；如果是多个片段，它就先把前面的部分拼成前缀，再枚举最后一个片段所有可能的组合。（这与我们原始的逻辑相同）生成的猜测会被收集到本地数组中，然后汇总到全局变量 guesses 里。

需要注意的是每个 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。

处理完毕之后，程序会为每个新生成的 PT 重新计算它的概率，然后按照概率从高到低插入回优先队列中。这一步是关键，因为它保证了系统总是优先处理概率最大的候选路径。原来那批已经处理完的 PT 则会从队列中删掉。

整个流程通过 MPI 的 MPI_Allreduce 实现了不同进程之间的总猜测数量同步，这样可以让系统保持一致性。

### 4.2.2 正确性验证

```
int pt_batch_size = size; // 可以根据需要调整
q.PopNextBatch_mpi(pt_batch_size);
```

修改 main.cpp (correctness_guess.cpp)，大小与进程数量匹配，并调用 PopNextBatch_mpi 函数，在 correctness_guess.cpp 运行了之后，结果如下图所示：

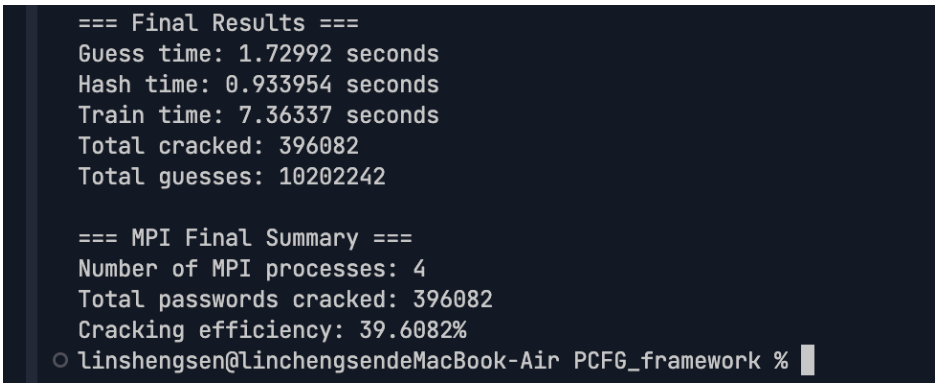


图 4.6: 多 pt 正确性验证

可以看到猜测对的口令数量与串行算法大致相同，说明了我们的多 pt 算法的正确性。

### 4.2.3 结果分析

我们重新对 main.cpp 进行编译，选取 30000000 这个数据规模，O2 编译下测试结果如下：

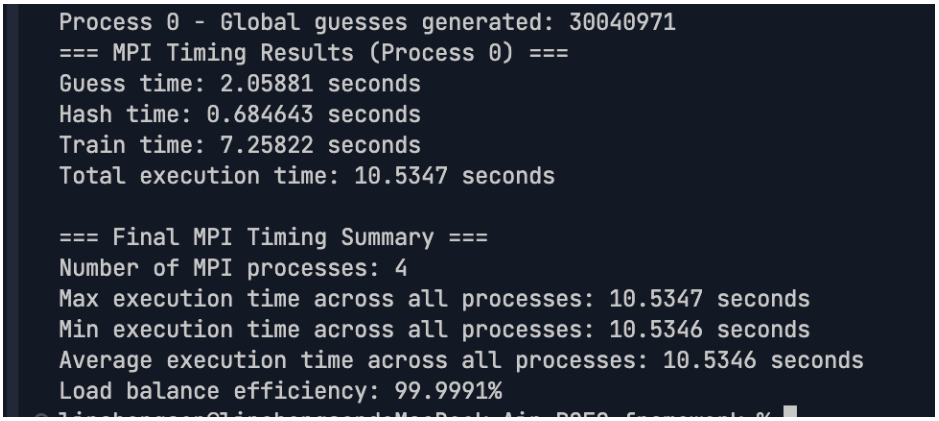


图 4.7: 多 pt 测试

可以看到，原来的 hash time 为 0.802938s，guess time 为 0.347614s，与我们之前的模式相比，hash time 得到了加速，guess time 变得慢了很多。

其中，guess time 的增长，我认为主要是因为这时候算法的工作模式发生了根本性的改变。原来的方案是每次只处理一个 PT，现在改成了批量处理多个 PT，这就带来了额外的进程管理开销。具体来说就是，程序需要先从优先队列中一次性取出多个 PT，然后把这些 PT 分配给不同的进程去处理，这个分发过程本身就需要时间。另外，由于我们采用了 MPI 并行计算，各个进程之间需要进行通信和同步等，必须要等所有进程都完成自己负责的 PT 处理后才能进行下一轮，否则会出错，这种同步等待也会增加时间开销。再加上批量处理需要更复杂的数据结构来管理这些 PT，内存分配和释放的操作也比之前频繁，所以整体的 guess time 就增加了很多。

hash time 的减小则体现了并行计算的优势。在新的实现中，多个 PT 可以同时生成密码，这意味着哈希计算的工作被分散到了多个进程中并行执行，而不是像之前那样串行地一个一个处理。这种并行处理大大提高了 CPU 的利用率，原本需要排队等待的 MD5 哈希计算现在可以同时进行。另外，批量处理还带来了一个意外的好处，就是提高了 CPU 缓存的命中率。当程序连续处理相似的数据时，CPU 缓存能够更有效地工作，减少了从内存中读取数据的次数。同时，由于工作负载被分散到多个进程，每个进程承担的哈希计算量也相对减少，单个进程的压力变小了，这也有助于提高整体的计算效率。虽然总的计算量没有减少，但是通过并行处理，我们实现了更高的吞吐量，所以 hash time 相应地就减少了。

### 4.3 修改进程数量

我尝试修改进程数量，从而来看不同进程数对并行算法的适配度。分别尝试使用 1，2，4，8 进程数。运行命令如下：(O2 1000w 数据规模)

```
mpirun -np 4 ./test_mpi_optimized
```

运行结果如下：

| 进程数 | hash time | guess time |
|---|---|---|
| 1 | 0.998997s | 0.138885s |
| 2 | 0.492738s | 0.119819s |
| 4 | 0.261684s | 0.106984s |
| 8 | 0.181863s | 0.376261s |

表 4: 不同进程数对比

可以看到基本上在进程数为 4 的情况下，hash time 和 guess time 的效果最好。或多或少都不好，我觉得主要存在以下的原因：

首先，进程数太少的时候，比如只有 1-2 个进程，并行度不够，CPU 资源没有充分利用，大部分计算任务还是集中在少数进程上，所以加速效果不明显。

其次，如果进程数太多，比如我们测试点 8 个进程，虽然理论上并行度更高，但是会带来额外的开销。每个进程都需要分配内存、维护自己的数据结构，而且进程间的通信成本也会大大增加。特别是我们代码中的 MPI_Allreduce 操作，进程越多，同步的时间就越长。另外，当进程数超过 CPU 核心数时，操作系统还需要进行进程调度，这也会产生额外的开销。

而当我们的进程数为 4 的时候正好达到了一个平衡点：既有足够的并行度来充分利用多核 CPU，又不会产生过多的通信和同步开销。

总的来说，并行计算不是进程越多越好，而是要在并行度和开销之间找到最佳平衡点。

# 5 心得体会

1. 通过这次实验，我对于多进程编程有了更深刻的认识，学会了如何基于 mpi 实现多进程编程。主要来说就是，MPI 多进程编程是一种基于消息传递的并行计算模式，其基本流程是首先通过 MPI_Init() 初始化运行环境并获取进程信息（命令行参数），然后根据进程数量将计算任务合理分解分配给各个进程，接着各进程通过 MPI_Send()、MPI_Recv() 等函数进行数据交换和通信协调，在计算过程中使用 MPI_Barrier() 等同步函数确保进程间的协调一致，最后将各进程的计算结果汇总到主进程 rank 0 并调用 MPI_Finalize() 清理资源。在本项目中，我们主要在 PT 层面实现了并行处理，通过批量处理优先队列中的多个 PT 来提高口令生成效率，同时在 MD5 哈希计算阶段利用 MPI 进行并行加速，从而显著提升了整体的计算性能。

2. 对于多进程编程，也是和我们上次的多线程编程一样，并不是进程数量越多越好，太多了会带来额外的开销，太少的话并行度不够，没有充分利用 cpu 资源。所以我们要在并行度和开销之间找到最佳平衡点。

3. 另外我对多 pt 并行算法进行了实现，其在一定程度上体现出了并行优势，但由于期末时间不是很够，所以并没有对其进行更进一步的优化。

# 6 代码链接

点击访问 GitHub 主页