南開大學
Nankai University

计算机学院

并行程序设计实验报告

基于 MPI 多进程的口令猜测优化

姓名：葛明宇

学号：2312388

专业：计算机科学与技术

2025 年 6 月 14 日

目录

1 引言 **2**

2 基础部分 **2**
2.1 并行设计 . . . 2
2.2 MPI 多进程实现 . . . 2
2.3 正确性验证 . . . 4
2.4 性能测试 . . . 6
2.5 结果分析 . . . 6
2.6 进程数对性能的影响 . . . 7
2.6.1 性能测试 . . . 7
2.6.2 结果分析 . . . 7

3 多 PT 并行优化 **7**
3.1 并行设计 . . . 7
3.1.1 任务分发 . . . 8
3.1.2 并行处理 . . . 8
3.1.3 结果聚合 . . . 8
3.2 MPI 多进程实现 . . . 8
3.3 性能测试 . . . 10
3.4 结果分析 . . . 11
3.5 进程数对性能的影响 . . . 11
3.5.1 性能测试 . . . 11
3.6 批次数对于多 PT 并行优化的影响 . . . 11
3.6.1 性能测试 . . . 11
3.6.2 结果分析 . . . 12

4 类似 Pipeline 的优化 **12**
4.1 并行设计 . . . 12
4.2 MPI 多进程实现 . . . 12
4.3 性能测试与对比 . . . 15
4.4 结果分析 . . . 15
4.5 优化方向 . . . 15

5 总结 **16**
5.1 优化策略 . . . 16
5.2 实验结论 . . . 16
5.3 未来工作 . . . 17

6 代码仓库 **17**

1 引言

在现代密码学领域，密码猜测攻击作为一种基础且有效的攻击方式，其效率提升始终是研究热点。随着高性能计算技术的发展，并行计算在密码破解场景中的应用愈发重要。在前期多线程实验中，我们已通过 pthread 和 OpenMP 实现了单个 Preterminal（PT）生成密码猜测时的内部并行化，通过阈值机制动态切换串行与并行算法。本次实验在此基础上，基于 MPI（Message Passing Interface）多进程模型，进一步对密码猜测过程进行优化，主要包括单个 PT 生成过程的并行化、多 PT 并发处理以及流水线式的口令生成与哈希计算协同优化，旨在探索并行程序设计在密码学中的应用潜力并提升攻击效率。

2 基础部分

2.1 并行设计

本实验的基础并行设计围绕 Generate() 函数展开，该函数负责将优先队列中弹出的”preterminal”生成具体口令并存储至全局 guesses 容器。核心优化点在于将生成过程的循环结构改造为多进程并行模式：通过将 preterminal 生成的口令值均分为 size 个数据块，每个进程独立处理一块数据，最终汇总结果。这种设计的关键优势在于：

(1) 负载均衡。通过动态计算每个进程的任务区间，确保前 remainder 个进程多分配 1 个任务，后续进程分配标准任务量，避免进程间负载失衡。

(2) 减少全局访问。各进程维护本地猜测列表，仅在最终阶段合并至全局容器，降低并发访问冲突。

2.2 MPI 多进程实现

在 PriorityQueue 类中新增 Generate_mpi(PT pt) 函数，其实现流程如下：

MPI 环境初始化：获取当前进程在通信域中的身份标识和通信域的总进程数。

```
//guessing_mpi.cpp
void PriorityQueue::Generate_mpi(PT pt)
{
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
};
```

由于在 Generate() 中的两个循环并行的的实现相差不大，这里给出 if 语句内部的完整实现，else 语句内部只展示不同的部分。

获取当前密码段的最大索引值，即该段可生成的密码总数，并计算每个进程的基础任务量。

```
//guessing_mpi.cpp
int total = pt.max_indices[0];
int base_chunk = total / size;
int remainder = total % size;
int start_idx, end_idx;
```

负载均衡的处理，前 remainder 个进程多分配一个任务，后面的进程分配 base_chunk 个任务。

```cpp
//guessing_mpi.cpp
if (rank < remainder) {
    start_idx = rank * (base_chunk + 1);
    end_idx = start_idx + base_chunk + 1;
} else {
    start_idx = remainder * (base_chunk + 1) + (rank - remainder) * base_chunk;
    end_idx = start_idx + base_chunk;
}
```

根据分配的索引区间生成密码猜测，并统计当前进程处理的任务数量。

```cpp
//guessing_mpi.cpp
for (int i = start_idx; i < end_idx; i++) {
    guesses.push_back(a->ordered_values[i]);
}
int local_count = (end_idx - start_idx);
```

使用 MPI_Allreduce() 函数将所有进程同时发送 local_count 到通信域，MPI 框架将所有进程的 local_count 求和, 并将求和结果 global_total 广播给所有进程。最后，将全局任务总数同步到类成员变量 total_guesses。

```cpp
//guessing_mpi.cpp
int global_total = 0;
MPI_Allreduce(&local_count, &global_total, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
total_guesses = global_total;
```

else 内部与 if 不同在于 guesses 容器最后接收的是前面所有 segment 合并的 value 与最后一个 segment 所有 value 的组合，而不仅仅是最后一个 segment 的所有 value。

```cpp
//guessing_mpi.cpp
string prefix;
• • •
for (int i = start_idx; i < end_idx; i++) {
    guesses.push_back(prefix + a->ordered_values[i]);
}
```

对于 main.cpp 也需要进行修改，修改后的 main_mpi.cpp 如下：

MPI_Init(&argc,&argv):初始化 MPI 环境,所有 MPI 程序必须在开始时调用此函数。MPI_Comm_rank(MPI_C 获取当前进程的编号（rank）,用于区分不同的进程。MPI_Comm_size(MPI_COMM_WORLD, &size)：获取 MPI 进程的总数（size）。

```cpp
    //main_mpi.cpp
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

MPI_Wtime()：返回一个以秒为单位的浮点数，表示从某个固定时间点开始到现在的时间。

```cpp
//main_mpi.cpp
double mpi_time_start_total = MPI_Wtime();  // 总体开始时间
double mpi_time_train_start = MPI_Wtime();
// ... 训练代码 ...
double mpi_time_train_end = MPI_Wtime();
time_train = mpi_time_train_end - mpi_time_train_start;
```

只有主进程(rank 为 0)会输出训练开始和结束的信息,避免多个进程重复输出。MPI_Barrier(MPI_COMM_WO
保证了所有进程在该函数处同步，确保所有进程都完成了训练阶段，再继续执行后续代码。

```cpp
//main_mpi.cpp

if (rank == 0) {
     cout << "Starting model training..." << endl;
}
if (rank == 0) {
    cout << "Model training completed in " << time_train << " seconds" << endl;
}
MPI_Barrier(MPI_COMM_WORLD);
```

将所有进程的局部猜测数 local_guesses 求和，得到全局猜测数 global_guesses。这样可以确保所有进程都能知道当前总共生成了多少个猜测。

```cpp
//main_mpi.cpp
int local_guesses = q.guesses.size();
int global_guesses = 0;
MPI_Allreduce(&local_guesses, &global_guesses, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
q.total_guesses = global_guesses;
```

一系列计时的统计。

```cpp
//main_mpi.cpp
double mpi_time_hash_start = MPI_Wtime();
// ... 哈希代码 ...
double mpi_time_hash_end = MPI_Wtime();
double hash_duration = mpi_time_hash_end - mpi_time_hash_start;
time_hash += hash_duration;
• • •
```

结束 MPI 环境，所有 MPI 程序必须在结束时调用此函数。

```cpp
//main_mpi.cpp
MPI_Finalize();
```

2.3 正确性验证

对 correctness.cpp 进行修改，具体修改如下:
初始化 MPI 环境，获取进程 ID 和总进程数，为并行计算做准备。

```
//correctness_mpi.cpp
MPI_Init(&argc, &argv);
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

仅让主进程 (rank 0) 输出训练开始信息，使用 MPI_Barrier 确保所有进程完成训练后再继续执行。

```
//correctness_mpi.cpp
if (rank == 0) {
    cout << "Starting model training with " << size << " MPI processes..." << endl;
}
q.m.train("input/Rockyou-singleLined-full.txt");
MPI_Barrier(MPI_COMM_WORLD);
```

仅主进程加载测试数据，然后通过 MPI_Bcast 广播到所有进程，所有进程都拥有完整测试集，用于后续的密码匹配验证。

```
//correctness_mpi.cpp
unordered_set<std::string> test_set;
vector<string> test_passwords;

if (rank == 0) {
    // ...省略加载代码...
}

int test_size = test_passwords.size();
MPI_Bcast(&test_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

for (int i = 0; i < test_size; i++) {
    int pw_len = 0;
    MPI_Bcast(&pw_len, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&test_passwords[i][0], pw_len, MPI_CHAR, 0, MPI_COMM_WORLD);
    if (rank != 0) {
        test_set.insert(test_passwords[i]);
    }
}
```

使用 MPI_Allreduce 汇总所有进程生成的口令数量，全局统计所有进程生成的口令总数，而不仅仅是本地进程。

```
//correctness_mpi.cpp
q.PopNext();
int local_guesses = q.guesses.size();
int global_guesses = 0;
MPI_Allreduce(&local_guesses, &global_guesses, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
q.total_guesses = global_guesses;
```

增加密码匹配逻辑，统计成功破解的密码数量，使用 MPI_Allreduce 汇总所有进程的破解计数，得到全局破解数量。

```
//correctness_mpi.cpp
for (const string& pw : q.guesses) {
    if (test_set.find(pw) != test_set.end()) {
        local_cracked++;  // 本地破解计数
    }
    MD5Hash(pw, state);
}

int current_global_cracked = 0;
MPI_Allreduce(&local_cracked, &current_global_cracked, 1, MPI_INT, MPI_SUM,
    MPI_COMM_WORLD);
```

最后仅主进程输出最终结果，避免多进程重复输出。

结果显示 cracked 的值与串行版本的结果一致，说明 MPI 多进程版本的正确性验证通过。

2.4 性能测试

本次测试的环境并不是实验提交的平台，这是由于平台的测试结果稳定以及当服务器处于高峰时需要进行任务的等待。那么，本次实验的测试是在“SCNet 国家超算互联网”平台进行的。当然，在实验提交的平台也有我相应的提交记录，并且结果也能大致反映性能测试的结论。

MPI 多进程版本使用的是 8 进程版本。

编译选项	串行 Guess 时间 (s)	MPI 并行 Guess 时间 (s)	加速比
不编译优化	12.979	13.190	0.984
O1 优化	0.561	0.275	2.040
O2 优化	0.581	0.295	1.969

表 1: MPI 并行优化加速比

在不进行编译优化时，MPI 多进程版本比串行时间还慢。在 O1 和 O2 优化时，MPI 多进程版本实现了加速，且 Guess 速度明显优于串行版本，加速比约为 2。

2.5 结果分析

(1) 编译优化的关键作用：未开启编译优化时，MPI 版本性能反而不如串行版本，O1 和 O2 优化下，MPI 版本实现了约 2 倍的加速比。这表明编译器优化有效减少了并行通信开销。

(2) 负载均衡的功效：代码采用了动态计算每个进程的任务区间，确保前 remainder 个进程多分配 1 个任务，后续进程分配标准任务量。从最后对于“Load balance efficiency”的统计结果超过 99.99%，说明负载均衡的设计起到了效果。

(3) 并行开销的影响：不优化时，进程间通信和同步开销超过了并行计算的收益；优化后，通信开销被显著降低，并行计算的优势得以体现。

(4) 加速比分析：使用 8 个进程仅获得约 2 倍加速比，未达到理想的 8 倍。这是由于口令生成任务本身的计算复杂度不高，通信开销占比相对较大；同时，全局变量访问和同步操作引入了额外开销。

2.6 进程数对性能的影响

选择对优化效果最好的 O1 进行进程数对性能影响的实验。

2.6.1 性能测试

进程数	2 进程	4 进程	8 进程	16 进程
加速比	1.705	2.133	2.040	0.019

表 2: 不同进程数加速比

从测试结果可以看出：在 2 进程、4 进程和 8 进程时，加速比逐渐增加，但 4 进程的加速比最高，为 2.133。在 16 进程时，加速比急剧下降，仅为 0.019，表明性能反而下降。

2.6.2 结果分析

(1) 并行开销：当进程数增加时，并行开销（如进程间通信、同步等）也随之增加。在进程数较少时，这些开销相对较小，因此加速比随着进程数的增加而增加。但是当进程数过多时，这些并行开销可能会超过并行带来的性能提升，导致加速下降比。

(2) 资源竞争：多个进程同时运行时，系统资源（如 CPU、内存等）可能会出现竞争。在进程数较少时，资源竞争相对较小，进程可以充分利用系统资源。但进程数过多时，资源竞争加剧，可能导致部分进程无法及时获取所需资源，从而影响整体性能。

从 8 进程比 4 进程慢看出 4 进程更适合完成这项任务, 对于并行带来的收益以及并行开销的平衡更优。

不过更值得分析的是 16 进程下的性能急速下降，加速比仅为 0.019。搜索资料后，我认为是以下的原因:

(1) 任务划分逻辑中，当进程数 $p = 16$ 时：若口令段总长度 total 较小，会出现任务碎片化（单个进程分配到的任务量接近 0）进程频繁处于空闲等待状态，而通信同步仍持续消耗资源，导致整体效率暴跌。

(2) 多进程共享计算节点内存带宽时,16 进程同时访问 guesses 容器、共享数据段（如 a->ordered_values），引发内存带宽争用。尤其在 emplace_back 操作时，大量进程并发写内存可能触发缓存一致性风暴，大幅降低内存访问效率。

(3) 在“SCNet 国家超算互联网”平台中,16 进程可能已占满节点核心，或触发跨节点通信（若进程调度到不同节点），跨节点通信延迟进一步恶化性能。

3 多 PT 并行优化

3.1 并行设计

处理多个 PT 时采用层次化的并行策略，将计算任务分解为三个主要阶段：批量获取、并行处理和结果聚合。

3.1.1 任务分发

主进程从优先队列中批量弹出多个 PT，并采用固定批量大小控制每次处理的任务量。同时，动态调整实际处理的 PT 数量，避免队列过小时的无效操作。

3.1.2 并行处理

将 PT 批次按进程数动态分配，确保负载均衡，同时，每个进程独立处理分配的 PT，生成局部猜测列表，并维护独立的局部猜测列表，避免频繁全局访问。

3.1.3 结果聚合

对于猜测聚合,每个进程将本地生成的猜测直接追加到全局容器,对于计数聚合,使用 MPI_Allreduce 聚合各进程的猜测计数，更新全局总数。最后所有进程还需要同步生成新 PT，计算概率后插入优先队列。采用有序插入策略，确保队列按概率降序排列。

3.2 MPI 多进程实现

对于代码的修改不再局限于 Generate() 函数，需要对一些相关接口也进行处理。

对 PopNext() 的修改

在 PopNextBatch_mpi() 中需要一次取出多个 PT，分配给各个进程，并进行最终的队列维护。获取当前进程的全局排名 rank 和总进程数 size，初始化 MPI 环境，计算实际要处理的 PT 数量，取用户请求的 batch_size 和队列当前大小的较小值，防止越界访问，从优先队列中取出前 actual_batch_size 个 PT，组成一个批次用于后续处理。这确保了每次处理的都是队列中优先级最高的任务。

```
//guessing_mpi_opt.cpp
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

int actual_batch_size = min(batch_size, (int)priority.size());
if (actual_batch_size == 0) return;

vector<PT> pt_batch;
for (int i = 0; i < actual_batch_size; i++) {
    pt_batch.push_back(priority[i]);
}
```

调用核心并行处理函数，将当前批次的 PT 分发给各个进程进行并行处理。

```
//guessing_mpi_opt.cpp
ProcessPTBatch_mpi(pt_batch);
```

遍历当前批次的每个 PT，调用 NewPTs() 方法生成新的 PT，并计算它们的概率值，将所有新 PT 收集到 all_new_pts 容器中。同时，从优先队列中删除已处理的批次，释放内存空间。最后，将新生成的 PT 按概率值插入到优先队列的适当位置，保持队列的降序排列。如果新 PT 的概率高于队列中某个位置的 PT，则插入该位置，否则添加到队列末尾。

```cpp
//guessing_mpi_opt.cpp
vector<PT> all_new_pts;
for (int i = 0; i < actual_batch_size; i++) {
    vector<PT> new_pts = priority[i].NewPTs();
    for (PT& pt : new_pts) {
        CalProb(pt);
        all_new_pts.push_back(pt);
    }
}

priority.erase(priority.begin(), priority.begin() + actual_batch_size);

for (PT& pt : all_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);
    }
}
```

核心并行处理函数 ProcessPTBatch_mpi() 的实现

获取进程基本信息, 并计算任务分配参数。

```cpp
//guessing_mpi_opt.cpp
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

int batch_size = pt_batch.size();

int pts_per_process = batch_size / size;
int remainder = batch_size % size;
```

进行任务区间计算，前 remainder 个进程会多分配 1 个任务（pts_per_process + 1）剩余进程分配标准任务量 pts_per_process 这种分配方式确保余数 remainder 被均匀分配给前 remainder 个进程。

```cpp
//guessing_mpi_opt.cpp
int start_pt, end_pt;
if (rank < remainder) {
    start_pt = rank * (pts_per_process + 1);
    end_pt = start_pt + pts_per_process + 1;
} else {
```

```
        start_pt = remainder * (pts_per_process + 1) + (rank - remainder) *
            pts_per_process;
        end_pt = start_pt + pts_per_process;
    }
```

local_guesses 存储当前进程生成的所有猜测口令 local_total_guesses 统计本地生成的口令总数这种设计避免了多进程直接操作全局容器。同时，遍历分配给当前进程的所有 PT（start_pt 到 end_pt）对每个 PT 调用 GenerateSinglePT_mpi 生成对应口令将生成的口令批量插入本地容器，而非逐次操作。

```
    //guessing_mpi_opt.cpp
    vector<string> local_guesses;
    int local_total_guesses = 0;

    for (int pt_idx = start_pt; pt_idx < end_pt; pt_idx++) {
        PT& pt = pt_batch[pt_idx];

        vector<string> pt_guesses;
        GenerateSinglePT_mpi(pt, pt_guesses);

        local_guesses.insert(local_guesses.end(), pt_guesses.begin(),
            pt_guesses.end());
        local_total_guesses += pt_guesses.size();
    }
```

将本地生成的所有猜测口令一次性合并到全局容器 guesses, 使用 insert 的范围版本，比多次插入更高效。并使用 MPI_Allreduce 实现高效的全局求和该操作将所有进程的 local_total_guesses 求和并返回给每个进程。

```
    //guessing_mpi_opt.cpp
    guesses.insert(guesses.end(), local_guesses.begin(), local_guesses.end());

    int global_total = 0;
    MPI_Allreduce(&local_total_guesses, &global_total, 1, MPI_INT, MPI_SUM,
        MPI_COMM_WORLD);
    total_guesses += global_total;
```

GenerateSinglePT_mpi() 函数

此函数负责为单个 PT 生成所有可能的口令组合，与串行版本 Generate() 函数相比只是去掉了生成口令数量的统计。这里便不再粘贴代码。

3.3 性能测试

这里的多 PT 执行时也是分配了 8 个进程。

编译选项	串行 Guess 时间 (s)	多 PT 并行 Guess 时间 (s)	加速比
不编译优化	12.979	104.297	0.124
O1 优化	0.561	2.628	0.213
O2 优化	0.581	2.849	0.204

表 3: 多 PT 并行优化加速比

无论选择什么编译优化，发现均会比串行版本慢很多，即使在开启了 O1 和 O2 优化的情况下，性能还是慢了接近 4 倍。

3.4 结果分析

对于上述结果，我猜测是因为会出现很多次剩余不足 8 进程的情况，最糟是队列里的 PT 数为 1 的情况。于是便思考着修改进程数，尝试减小进程数以削弱上述情况的影响。

3.5 进程数对性能的影响

选择优化效果最好的 O1 优化下进行进程数对性能影响的实验。

3.5.1 性能测试

进程数	2 进程	4 进程	8 进程
加速比	0.133	0.162	0.213

表 4: 多 PT 不同进程数加速比

可结果显示 2、4、8 线程中，线程数越大，加速比越大。既然不是上面分析的原因，那么到底是哪里的问题？于是我又开始了探索。

3.6 批次数对于多 PT 并行优化的影响

这里的批次数 batch_size 是指每次 PopNextBatch_mpi() 函数需要处理的 PT 数，那么每个进程也就要处理 batch_size/size 个 PT，size 是进程数。

在探索的过程中，最终发现，是在主函数中的 PopNextBatch_mpi() 函数调用时，批次数为进程数，那么在 PopNextBatch_mpi() 的任务分配中，每个进程每次只处理一个 PT，导致了多进程资源开销过大，于是便无法实现加速。

3.6.1 性能测试

经过我的测试，在进程数为 8 的情况下，加速比结果汇总如下 (表格中批次倍数表示 size 的倍数)：

批次倍数	25*	50*	75*	100*
加速比	1.565	1.12	0.747	0.563

表 5: 不同批次数加速比

在我选取的几个批次倍数的样例中，当批次倍数为 25 时，加速效果最好，加速比能够达到 1.565。反而批次倍数继续增大，加速比会越来越低。

3.6.2 结果分析

(1) 批次数过小的影响

当批次数过小时，每个进程每次处理的 PT 数量较少。在进程数为 8 的情况下，如果批次数仅为进程数，即每个进程每次只处理一个 PT，这会导致多进程资源开销过大，无法实现加速。

具体而言，频繁地启动和切换进程会消耗大量的系统资源，包括 CPU 时间和内存。每个进程在处理一个 PT 后，需要进行上下文切换，将控制权交给其他进程，这个过程会产生额外的开销。此外，进程之间的同步和通信也会变得更加频繁，进一步增加了系统的负担。因此，批次数过小使得进程的优势没有得到充分发挥，系统性能反而受到影响。

(2) 批次数适中的优势

从测试结果来看，当批次倍数为 25 时，加速效果最好，加速比能够达到 1.565。这是因为在这个批次数下，每个进程能够处理相对合理数量的 PT，使得进程的并行计算能力得到了较好的发挥。进程在处理多个 PT 的过程中，可以减少上下文切换的次数，提高 CPU 的利用率。同时，适中的批次数也能够在一定程度上平衡进程之间的负载，避免某些进程空闲而其他进程繁忙的情况。这样，系统能够充分利用多进程的并行计算能力，从而实现较好的加速效果。

(3) 批次数过大的问题

随着批次倍数的继续增大，加速比会越来越低。当批次数过大时，虽然每个进程能够充分利用自身的资源，处理大量的 PT，但在结果的汇总过程中会产生大量的通信开销和维护队列的开销。

具体来说，在多进程并行计算中，每个进程处理完自己分配的 PT 后，需要将结果汇总到主进程或其他进程进行统一处理。批次数越大，每个进程处理的结果就越多，进程之间的通信量也会相应增加。频繁的通信会导致网络延迟和带宽占用，降低系统的整体性能。此外，维护优先队列也会变得更加复杂和耗时。在 PopNextBatch_mpi() 函数中，处理完一批 PT 后，需要将新生成的 PT 插入到优先队列中。批次数越大，新生成的 PT 数量就越多，插入操作的时间复杂度也会相应增加，这会进一步影响系统的性能。当然，批次数过大也会带来其他问题，比如内存占用过大，降低缓存命中率。

4 类似 Pipeline 的优化

4.1 并行设计

这里在原有 Generate() 函数的基础上，增加了 Pipeline 优化。将口令破解任务分解为两个主要阶段并分配给不同进程：

(1) 进程 0 作为口令生成器

基于 PCFG (概率上下文无关文法) 模型生成可能的口令猜测，按照概率优先级队列组织和生成口令，并将生成的口令批量发送给哈希计算进程。

(2) 进程 1 作为哈希计算器

接收来自口令生成进程的口令，对每个口令计算 MD5 哈希值，最后统计处理过的口令数量。

4.2 MPI 多进程实现

初始化 MPI 并行环境，获取当前进程数 (world_size) 和当前进程编号 (world_rank)。创建高精度时钟点用于性能计时，记录训练结束时间、处理开始时间和处理结束时间。同时定义用于存储训练

时间和总执行时间的变量。

```
//main_mpi_pipeline.cpp
MPI_Init(&argc, &argv);

int world_size, world_rank;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

time_point<high_resolution_clock> train_end_time;
time_point<high_resolution_clock> process_start_time;
time_point<high_resolution_clock> process_end_time;

double train_time = 0.0;
double total_time = 0.0;
```

确保程序有足够的进程执行流水线并行任务（1 个生成口令，1 个计算哈希）。

```
//main_mpi_pipeline.cpp
if (world_size < 2) {
    cerr << "This application requires at least 2 MPI processes." << endl;
    MPI_Abort(MPI_COMM_WORLD, 1);
}
```

作为整个破解任务的终止条件，当生成的口令数达到该值时停止生成。

```
//main_mpi_pipeline.cpp
const int MAX_GUESSES = 10000000;
```

进程 0（口令生成器）核心逻辑:

当前进程作为口令生成器（world_rank == 0），使用 Rockyou 数据集训练 PCFG 模型，计算口令概率分布。中间部分与串行版本相同。口令生成完成之后，需要先发送批量大小，再逐个发送口令长度和内容 (这种实现最简单，但是非常耗费时间)，记录训练时间和总执行时间，输出生成的口令总数。并发送终止信号 (-1) 通知哈希计算进程结束任务。

```
//main_mpi_pipeline.cpp
if (world_rank == 0) {
    • • •
    while (!q.priority.empty() && total_guesses < MAX_GUESSES) {
        • • •
        if (total_guesses + q.guesses.size() >= MAX_GUESSES) {
            total_guesses = MAX_GUESSES;
            break;
        }
        else if (q.guesses.size() >= 1000000) {
            int batch_size = q.guesses.size();
            MPI_Send(&batch_size, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

            for (string& pw : q.guesses) {
```

```
                int len = pw.size();
                MPI_Send(&len, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
                MPI_Send(pw.c_str(), len, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
            }

            total_guesses += batch_size;
            q.guesses.clear();
        }
    }
    ...
    int end_signal = -1;
    MPI_Send(&end_signal, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
}
```

进程 1（哈希计算器）核心逻辑:

当前进程作为哈希计算器（world_rank == 1），持续接收来自进程 0 的口令批量。先接收批量大小，再逐个接收口令长度和内容，并动态分配缓冲区存储口令，处理后立即释放。使用 MD5Hash 函数计算每个口令的哈希值后，当接收到终止信号 (-1) 时退出循环。

```
//main_mpi_pipeline.cpp
else if (world_rank == 1) {
    int total_hashes = 0;

    while (true) {
        int batch_size;
        MPI_Recv(&batch_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);

        if (batch_size == -1) break;

        for (int i = 0; i < batch_size; i++) {
            int len;
            MPI_Recv(&len, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            char* buffer = new char[len + 1];
            MPI_Recv(buffer, len, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
            buffer[len] = '\0';
            string pw(buffer);
            delete[] buffer;

            bit32 state[4];
            MD5Hash(pw, state);
        }
        total_hashes += batch_size;
    }
    cout << "Total hashes computed: " << total_hashes << endl;
}
```

调用 MPI_Finalize () 清理 MPI 环境资源。

```
//main_mpi_pipeline.cpp
MPI_Finalize();
```

4.3 性能测试与对比

在这一部分我们关注的是除了训练时间外，口令生成和哈希计算的总时间，在串行源码和我的类似 pipeline 的优化版本上都进行了这一部分的统计。最终结果如下：

编译选项	串行 Guess+Hash 时间 (s)	MPI 并行 Guess+Hash 时间 (s)	加速比
不编译优化	23.6004	30.9829	0.762
O1 优化	3.51221	7.4282	0.473
O2 优化	3.52399	7.4332	0.474

表 6: pipeline 优化加速比

无论是哪种编译选项下，类似 pipeline 的口令生成与哈希优化均无法实现加速。

4.4 结果分析

尽管我们的实现方式采用了阻塞式通信机制，实现了类似流水线的操作，但根据实验结果，哈希操作在整体任务中占比较大，这在 Guess 时间和 Guess+Hash 时间的测试结果中得到了体现。

在哈希计算过程中，口令生成器可以继续执行下一轮口令生成任务，但由于通信机制的限制，当新一轮口令生成完成后，必须等待当前哈希操作结束才能将新生成的口令传输给哈希进程。这意味着，只有在哈希进程成功接收口令后，口令生成器才能开展新一轮的执行工作。正是由于两个进程之间的通信消耗远远超过了口令生成所需的时间，导致了流水线的效率低下，失去了流水线优化的初衷。

此外，在哈希操作执行期间，由于无法进行进程间的信息传输，流水线会被阻塞，使得口令生成进程需要长时间等待哈希进程开始接收新的口令，从而进一步加剧了流水线的效率问题。

综上所述，阻塞式通信机制的应用在流水线并行优化过程中存在明显的性能瓶颈，主要表现为通信时间过长以及流水线阻塞，这两方面问题共同导致了流水线效率的低下，使得流水线优化的实际效果未能达到预期。

4.5 优化方向

(1) 添加缓冲区，口令生成由缓冲区负责传输给哈希进程。这样口令生成进程就不会因为等待哈希进程而阻塞。具体而言，缓冲区的引入可以在口令生成和哈希计算之间起到一个“中转站”的作用。当口令生成进程产生一批口令后，不是直接立即发送给哈希进程，而是先将它们暂存到缓冲区中。

(2) 优化口令的传输性能。可以考虑实现批次化的传输，减少通信的次数。批次化传输的基本思想是将多个口令打包成一个批次，然后一次性进行传输，而不是像之前那样逐个发送口令。这样可以显著减少通信的次数，从而降低通信开销对整体性能的影响。

5 总结

在本次实验中，我们系统地探索了基于 MPI 的并行优化技术在 PCFG（概率上下文无关文法）口令破解中的应用。通过三个层次的优化策略，我们深入研究了如何利用并行计算来提高口令破解的效率。

5.1 优化策略

(1) 单个 PT 的多进程并行化：

在基础部分，我们实现了单个 PT 内 Generate() 函数的多进程并行化。通过将密码生成任务划分为多个子任务，并分配给不同的进程处理，我们显著提高了密码生成的效率。实验结果表明，在开启编译器优化（O1 和 O2）的情况下，多进程并行化能够实现大约两倍的加速比。然而，未开启编译器优化时，由于并行通信开销较大，并行版本的性能反而不如串行版本。

(2) 多 PT 并行优化：

在多 PT 并行优化中，我们将并行策略扩展到同时处理多个 PT。通过层次化的并行策略，我们实现了任务的批量获取、并行处理和结果聚合。实验结果表明，适当的批次数能够显著提高并行效率。当批次数设置为进程数的 25 倍时，加速比达到了 1.565。然而，批次数过大或过小都会导致性能下降。这表明，批次数的合理设置对于多 PT 并行优化至关重要。

(3) 类似 Pipeline 的优化：

在类似 Pipeline 的优化中，我们将口令破解任务分解为两个主要阶段：口令生成和哈希计算。通过使用两个进程分别负责这两个阶段，并在它们之间进行通信，我们尝试构建一个流水线式的并行处理流程。然而，实验结果表明，由于通信开销较大，流水线式的优化并未实现预期的性能提升。这主要是因为当前实现中采用的阻塞式通信机制导致了流水线阻塞，口令生成进程大量时间处于等待哈希进程接收新口令的状态。

5.2 实验结论

(1) 编译器优化的重要性：

实验结果表明，编译器优化（如 O1 和 O2）对于提高并行程序的效率至关重要。未开启编译器优化时，并行通信开销较大，导致并行版本性能不如串行版本。而开启优化后，通信开销显著降低，并行计算的优势得以体现。

(2) 进程数对性能的影响：

在多数场景下，4 进程和 8 进程配置能够达到最佳性能。过多的进程数（如 16 进程）会导致任务碎片化和内存带宽争用，从而降低整体效率。

(3) 批次数对多 PT 并行的影响：

批次数的合理设置对多 PT 并行优化具有重要影响。过小的批次数会导致进程资源未充分利用，而过大的批次数会增加通信开销和队列维护开销。实验表明，批次数设置为进程数的 25 倍时能够获得最佳加速比。

(4) 流水线优化的瓶颈：

当前实现的流水线优化由于采用阻塞式通信机制，导致通信开销较大，流水线阻塞现象严重。未来工作可以通过引入非阻塞通信和双缓冲技术来优化流水线性能。

5.3 未来工作

(1) 批次数动态调整机制：

实现自适应批次数调整算法，根据历史执行时间预测最佳批次数。

(2) 混合并行架构：

结合 MPI 与 OpenMP 实现两级并行。MPI 进程间并行处理不同 PT 批次。OpenMP 线程级并行处理单个 PT 内的密码生成。

(3) 通信优化策略：

采用 MPI 非阻塞通信 (Isend/Irecv) 实现计算与通信重叠，并设计高效的数据压缩算法，减少口令传输量

6 代码仓库

仓库链接 Gitee