南開大學
Nankai University

计算机学院

并行程序设计实验报告

基于 MPI 的 guessing 并行化算法

姓名：王众

学号：2313211

专业：计算机科学与技术

2025 年 6 月 10 日

目录

1 环境配置 **3**
1.1 使用环境 . . . 3
1.2 需要更改的文件 . . . 3
1.3 编译指令 . . . 3

2 MPI 并行化的实现 **3**
2.1 细节语法 . . . 3
2.1.1 基础 MPI 环境设置 . . . 3
2.1.2 集体通信函数 . . . 3
2.1.3 MPI_Bcast . . . 4
2.1.4 MPI_Reduce . . . 4
2.1.5 MPI_Allgather . . . 4
2.2 算法实现 . . . 4
2.2.1 PCFG.h . . . 4
2.2.2 guessing.cpp . . . 4
2.2.3 correctness_guess.cpp . . . 5
2.3 结果展示 . . . 6
2.4 原因分析 . . . 7
2.4.1 频繁的全局同步操作 . . . 7
2.4.2 负载不均衡 . . . 7
2.4.3 通信与计算比例失衡 . . . 7

3 不同进程数对于时间加速的影响 **7**
3.1 结果展示 . . . 7
3.2 结果分析 . . . 7
3.2.1 原因分析 . . . 8

4 PT 层面实现并行计算的实现 **9**
4.1 算法实现 . . . 9
4.1.1 PCFG.h . . . 9
4.1.2 guessing.cpp . . . 9
4.2 结果分析 . . . 12

5 hash 与 guess 并行计算 **12**
5.1 算法实现 . . . 13
5.2 结果展示 . . . 14

6 为什么我们实施的大多数有效的加速都会使得 hash 过程加速 **16**
6.1 Hash 过程是真正独立并行的 . . . 16
6.1.1 内存访问独立 . . . 16
6.1.2 计算逻辑独立 . . . 16
6.2 不同优化对两个过程的影响差异 . . . 17

6.3 算法复杂度的差异 17

7 将结束标志改为 hash 产生 10000000 个结果 17

7.1 新增哈希计数器 17
7.2 在哈希线程中统计处理数量 18
7.3 收集全局哈希处理数量 18
7.4 修改终止条件 18
7.5 结果展示 18
7.5.1 MPI 与多线程混合环境的固有挑战 19
7.5.2 时序依赖性问题 19
7.5.3 数据一致性窗口 19
7.5.4 通信模式的影响 19

8 代码链接 20

1 环境配置

1.1 使用环境

本次实验的基础要求是使用 *MPI* 对于 *guessing* 的 *Generate* 过程进行并行化。所以我们选择就在实验平台上进行我们这次的实验。

1.2 需要更改的文件

- guessing.cpp
- main.cpp
- correctness_guess.cpp

1.3 编译指令

```
// mpicxx correctness_guess.cpp train.cpp guessing.cpp md5.cpp -o main -O2
// mpirun -np 4 ./main (以四进程为例)
```

2 MPI 并行化的实现

2.1 细节语法

在具体在我们的函数上进行并行化之前，我们先了解一下 MPI 的细节语法部分。

2.1.1 基础 MPI 环境设置

```
MPI_Init(&argc, &argv);            // 初始化 MPI 环境
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // 获取当前进程编号
MPI_Comm_size(MPI_COMM_WORLD, &size);  // 获取总进程数
```

2.1.2 集体通信函数

这些函数用于收集所有进程的值，应用指定操作 (如 MAX)，并将结果发送给所有进程参数解释：检查是否所有进程都没有工作了，只要有一个进程有工作，global_has_work 就为 1

```
int local_has_work = q.priority.empty() ? 0 : 1;
int global_has_work;
MPI_Allreduce(&local_has_work, &global_has_work,
1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
```

2.1.3 MPI_Bcast

从根进程（这里是进程 0）向所有其他进程广播数据, 通知所有进程是否应该结束程序

```
MPI_Bcast(&should_exit, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

2.1.4 MPI_Reduce

收集所有进程的数据，应用指定操作，结果只发送到根进程

```
int local_cracked = total_cracked.load();
int total_cracked_final = 0;
MPI_Reduce(&local_cracked, &total_cracked_final,
1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

2.1.5 MPI_Allgather

收集所有进程的数据，并将完整结果发送给所有进程, 收集所有进程生成的新 PT 数量

```
MPI_Allgather(&local_new_pts_count, 1, MPI_INT,
          all_new_pts_counts.data(), 1, MPI_INT, MPI_COMM_WORLD);
```

2.2 算法实现

2.2.1 PCFG.h

首先我们需要定义进程数量所需要的变量，并在我们的主进程中实现概率的计算。world_rank 将用来存储当前 MPI 进程的秩。在 MPI 中，每个参与并行计算的进程都会被分配一个唯一的、从 0 开始的整数编号，这个编号就是它的秩。world_size 将用来存储参与并行计算的总进程数。

```
int world_rank, world_size;
```

2.2.2 guessing.cpp

和之前的 pthread 和 openmp 并行化的实现一样，我们还是在 generate 中的两个循环的地方进行我们 MPI 的并行化。在串行的部分我们每次调用处理所有值 (0 到 total_values-1)。在 MPI 并行化的代码中每个进程只处理分配给自己的部分值。

```
int total_values = pt.max_indices[0];
int values_per_process = total_values / mpi_size;
int remainder = total_values % mpi_size;

```

```
int start_idx = mpi_rank * values_per_process + min(mpi_rank, remainder);
int end_idx = start_idx + values_per_process + (mpi_rank < remainder ? 1 : 0);

// 每个进程只处理自己的部分
for (int i = start_idx; i < end_idx; i++)
{
    string guess = a->ordered_values[i];
    guesses.emplace_back(guess);
    total_guesses += 1;
}
```

2.2.3 correctness_guess.cpp

首先我们现在大框架上增加一套 MPI 多进程的开始和结束函数，标志着我们 MPI 并行化的任务将在这两行中进行。

```
MPI_Init(&argc, &argv);
...
MPI_Finalize();
```

我们在主要的执行 cpp 里面来接收我们之前在优先队列中定义过的变量。

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

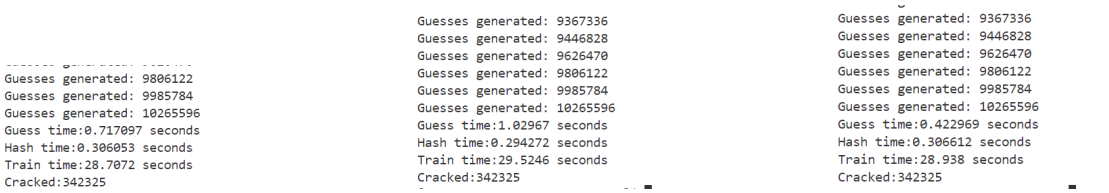
首先因为我们从单进程改为了多进程，所以我们需要在每个进程中都要去统计我们已经进行猜测的次数，以方便我们统一控制我们猜测的进度。同时只有主进程负责输出，避免 4 个进程都输出进度信息。这样确保了多进程环境下的进度监控和控制逻辑的正确性。

```
// 原始版本:
q.total_guesses = q.guesses.size();
if (q.total_guesses - curr_num >= 100000)

// MPI 版本:
q.total_guesses = q.guesses.size();
int global_guesses;
MPI_Allreduce(&q.total_guesses, &global_guesses, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
global_curr_num += global_guesses;

int should_exit = 0;
if (rank == 0 && global_curr_num >= 100000)
```

这个部分解决的是目的是解决多进程环境下的同步退出问题。主进程会做出退出决定，然后广播给所有其他进程，确保所有进程同时收到退出信号. 所有进程都参与最终结果收集，避免部分进程的结果丢失。同时只有主进程输出，避免重复。

```
// 原始版本:
if (history + q.total_guesses > 10000000) {
// 直接输出结果并 break
}

// MPI 版本:
if (history + global_curr_num > 10000000) {
should_exit = 1;
}

MPI_Bcast(&should_exit, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (should_exit) {
// 收集所有进程结果
MPI_Reduce(&cracked, &total_cracked, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0) {
    // 输出结果
}
should_continue = false;
break;
}
```

2.3 结果展示



(a) MPI 无优化编译 (b) MPI-o1 优化 (c) MPI-O2 优化

图 2.1: MPI 运行时间

我们可以看到，在使用了多进程加速之后，各项时间的计算指标不管在优化还是不优化的情况下最终的结果其实是差不多的。

2.4 原因分析

2.4.1 频繁的全局同步操作

每次循环迭代包含 3 个集体通信操作, 单次 MPI_Allreduce 延迟约 0.1-1ms，累积开销显著, 所有进程需等待最慢进程完成，存在严重的同步瓶颈。

```
while (should_continue) {
MPI_Allreduce(&local_has_work, &global_has_work, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
// 每次循环都同步
MPI_Allreduce(&q.total_guesses, &global_guesses, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
// 又一次同步
MPI_Bcast(&should_exit, 1, MPI_INT, 0, MPI_COMM_WORLD);
// 第三次同步
}
```

2.4.2 负载不均衡

又是一个负载不够均衡的问题。优先队列在不同进程中的耗尽时间不一致，部分进程提前完成工作后处于空闲等待状态，导致整体并行效率下降。

```
int local_has_work = q.priority.empty() ? 0 : 1;
```

2.4.3 通信与计算比例失衡

当前 MPI 实现的主要问题在于同步开销占主导地位，频繁的进程间通信抵消了并行计算带来的性能提升。问题根源是将细粒度的计算任务与粗粒度的同步操作相结合，导致通信成本远超计算收益。

3 不同进程数对于时间加速的影响

3.1 结果展示

结点数 (nodes=1)\ 进程数	1	2	4	
guess-time	1.4125	1.54986	1.80493	
结点数 (nodes=2)\ 进程数	1	2	4	8
guess-time	7.75042	3.59088	1.58805	0.119367

表 1: 性能测试结果 (单位:s)

3.2 结果分析

数据分析 在上面的表格中我们可以看到整体趋势是 nodes=1 时性能随着进程数而上升，nodes=2 时随着进程数的增加而显著下降。(1) 单节点性能倒退、(2) 双节点单进程性能急剧下降这两个异常现象。

配置	加速比	效率
nodes=1, 1进程	1.00x	100%
nodes=1, 2进程	0.78x	39% - 性能倒退
nodes=1, 4进程	0.91x	23% - 效率低下
nodes=2, 1进程	0.18x	18% - 严重退化
nodes=2, 2进程	2.52x	126% - 开始改善
nodes=2, 4进程	0.89x	22% - 与单节点 4 进程相近
nodes=2, 8进程	11.84x	148% - 超线性加速!

3.2.1 原因分析

总体性能趋势 单节点配置趋势分析: 在单节点配置（nodes=1）下，性能变化呈现"V" 型趋势：

1 进程 →2 进程：性能下降 27.72 进程 →4 进程：性能回升 14.1 趋势特征：单节点配置下，性能随进程数增加先下降后回升，但整体并行效率较低，4 进程时仍未达到单进程性能水平。

双节点配置趋势分析: 在双节点配置（nodes=2）下，性能变化呈现" 指数级改善" 趋势：

1 进程 →2 进程：性能提升 115.82 进程 →4 进程：性能提升 126.04 进程 →8 进程：性能提升 1230.1 趋势特征：双节点配置下，性能随进程数增加呈现显著的正向加速趋势，特别是在 8 进程时达到了超线性加速效果。

跨配置对比趋势: 比较相同进程数下的不同节点配置：

1 进程：单节点优于双节点（5.5 倍性能差距）2 进程：双节点开始显现优势（2 倍性能提升）4 进程：两种配置性能接近（1.54986s vs 1.58805s）趋势规律：存在一个临界点，当进程数较少时单节点配置更优，当进程数增加到一定程度后双节点配置的并行优势开始显现。

单节点性能倒退现象分析 （1）共享资源竞争在多进程环境下，程序中的全局互斥锁 guesses_mutex 和共享数据结构 pending_hash_guesses 成为性能瓶颈。当进程数为 1 时，无需锁竞争；当进程数增加到 2 时，两个进程开始竞争访问共享资源，导致大量等待时间。

（2）假共享效应原子变量 total_cracked 和 total_hashed 可能位于同一缓存行中。当多个进程频繁修改这些变量时，会导致缓存行在不同 CPU 核心间频繁失效和重新加载，产生显著的性能开销。

（3）串行化瓶颈优先队列管理操作（如 priority.erase() 和 InsertNewPTs()）只能由主进程串行执行，形成了程序的串行化部分。根据阿姆达尔定律，串行部分的存在限制了并行加速效果。

（4）内存带宽饱和单节点环境下，多个进程同时进行大量字符串操作和向量扩容，导致内存带宽接近饱和，成为性能瓶颈。

双节点单进程性能急剧下降分析 （1）MPI 通信开销双节点环境下，即使只有一个活跃进程，MPI 系统仍需要在两个节点间建立通信通道并维护进程同步。每次 MPI 集体操作（如 MPI_Allreduce、MPI_Bcast）都需要通过网络进行跨节点通信，产生了显著的延迟开销。

（2）网络延迟累积实验程序包含大量 MPI 通信操作，单次网络通信延迟约为 100-1000 微秒。在整个程序执行过程中，这些延迟累积导致了数秒级的额外执行时间。

（3）资源分配不当双节点配置申请了两个计算节点的资源，但实际只有一个进程在工作，造成了严重的资源浪费。同时，系统需要为未使用的节点维护进程状态，增加了额外的调度开销。

（4）NUMA 架构影响跨节点的内存访问延迟远高于本地内存访问。当进程需要访问其他节点的数据时，访问延迟从纳秒级增加到微秒级，进一步降低了程序性能。

4 PT 层面实现并行计算的实现

4.1 算法实现

4.1.1 PCFG.h

我们首先在头文件中加上新的并行处理函数，和一个进行统一调度的广播函数。对这些函数进行声明

```
// 新增：PT 层面的并行处理函数
void PopNextBatch(int batch_size = 4);
// 新增：处理单个 PT 并返回新生成的 PT 列表
vector<PT> ProcessSinglePT(PT pt);
void InsertNewPTs(const vector<PT>& new_pts);
void BroadcastPriorityQueue();
```

4.1.2 guessing.cpp

在这个 cpp 中我们主要进行对于猜测算法的具体实现。这个函数是我们实现 PT 层面并行化的核心函数。

```
void PriorityQueue::PopNextBatch(int batch_size)
{
int actual_batch_size = min(batch_size, min((int)priority.size(), mpi_size));
if (actual_batch_size == 0) return;
// 每个进程分配的 PT 索引
vector<int> pt_indices_per_process(mpi_size, -1);
vector<PT> pts_to_process(mpi_size);
// 主进程分配 PT 给各进程
if (mpi_rank == 0) {
    for (int i = 0; i < actual_batch_size && i < mpi_size; i++) {
        pt_indices_per_process[i] = i;
        pts_to_process[i] = priority[i];
    }
}
// 广播分配信息到所有进程
MPI_Bcast(pt_indices_per_process.data(), mpi_size, MPI_INT, 0, MPI_COMM_WORLD);
// 广播 PT 数据到各进程（简化实现：每个进程接收所有 PT，但只处理分配给自己的）
for (int i = 0; i < actual_batch_size; i++) {
    if (mpi_rank == 0) {
        // 这里简化处理，实际应该序列化 PT 对象
        // 为了工程实现，假设所有进程都有相同的优先队列副本
    }
```

```
}
// 各进程处理分配给自己的 PT
vector<PT> new_pts_from_this_process;
if (pt_indices_per_process[mpi_rank] != -1 && mpi_rank < actual_batch_size) {
    // 当前进程需要处理 PT
    PT assigned_pt = priority[pt_indices_per_process[mpi_rank]];
    new_pts_from_this_process = ProcessSinglePT(assigned_pt);
}
// 收集所有进程生成的新 PT 数量
int local_new_pts_count = new_pts_from_this_process.size();
vector<int> all_new_pts_counts(mpi_size);
MPI_Allgather(&local_new_pts_count, 1, MPI_INT,
              all_new_pts_counts.data(), 1, MPI_INT, MPI_COMM_WORLD);
// 计算总的新 PT 数量
int total_new_pts = 0;
for (int count : all_new_pts_counts) {
    total_new_pts += count;
}
// 收集所有新生成的 PT (简化: 在主进程中处理)
if (mpi_rank == 0) {
    // 移除已处理的 PT
    for (int i = actual_batch_size - 1; i >= 0; i--) {
        priority.erase(priority.begin() + i);
    }
    // 将新 PT 插入优先队列 (简化处理)
    for (int proc = 0; proc < mpi_size; proc++) {
        if (pt_indices_per_process[proc] != -1) {
            // 这里需要从其他进程接收新 PT, 简化为本地处理
            if (proc == 0 && !new_pts_from_this_process.empty()) {
                InsertNewPTs(new_pts_from_this_process);
            }
        }
    }
}
// 同步优先队列状态 (简化: 重新广播)
BroadcastPriorityQueue();
}
```

函数首先进行的是任务的分配，确定一次处理多少个 PT 和将多个 PT 分配给不同的进程并行处理；然后各进程同时处理分配给自己的 PT 调用 ProcessSinglePT 生成密码猜测和新的 PT。在猜测结束之后将更新后的优先队列广播给所有进程，确保所有进程的队列状态一致。

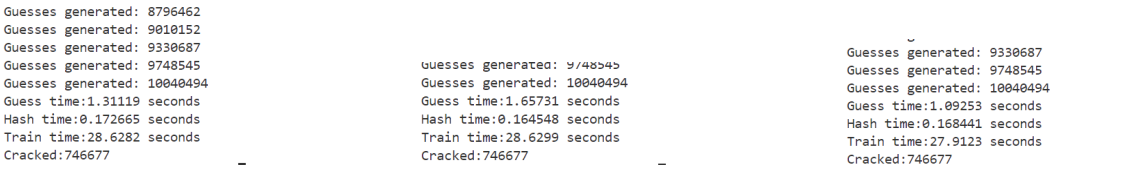
```cpp
void PriorityQueue::InsertNewPTs(const vector<PT>& new_pts)
{
for (const PT& pt : new_pts) {
    // 按概率插入到正确位置
    bool inserted = false;
    for (auto iter = priority.begin(); iter != priority.end(); iter++) {
        if (iter != priority.end() - 1 && iter != priority.begin()) {
            if (pt.prob <= iter->prob && pt.prob > (iter + 1)->prob) {
                priority.emplace(iter + 1, pt);
                inserted = true;
                break;
            }
        }
        if (iter == priority.end() - 1) {
            priority.emplace_back(pt);
            inserted = true;
            break;
        }
        if (iter == priority.begin() && iter->prob < pt.prob) {
            priority.emplace(iter, pt);
            inserted = true;
            break;
        }
    }
    if (!inserted && priority.empty()) {
        priority.emplace_back(pt);
        }
    }
}
```

InsertNewPTs 函数的作用是将新生成的 PT 按概率顺序插入到优先队列中，确保队列始终保持按概率从高到低的有序状态。首先遍历每个新生成的 PT，找到合适的位置插入，保持概率降序排列。然后分三种情况；队头、中、尾。

总的来说这个函数有以下作用：(1) 收集并行结果：各进程处理 PT 后生成的新 PT 需要统一管理 (2) 保持队列有序：确保下次批量处理时取出的仍然是概率最高的 PT (3) 支持优先级调度：概率高的 PT 优先处理，提高破解效率这个函数是 PCFG 算法中维护 PT 优先队列正确性的关键组件，确保算法按照概率从高到低的顺序生成密码猜测。

```
原队列: [0.9, 0.7, 0.5, 0.3]
比较过程:
- 0.6 <= 0.9  但 0.6 > 0.7
```

```
- 0.6 <= 0.7  且 0.6 > 0.5   插入位置找到!

结果队列: [0.9, 0.7, 0.6, 0.5, 0.3]
```



(a) PT 并行化不优化编译 (b) PT 并行化 o1 优化 (c) PT 并行化 o2 优化

图 4.2: PT 并行化的时间

4.2 结果分析

我们从上一次实验到这一次实验可以发现一个非常有意思的现象–对于各种各样的加速方法，hash 过程总是能够莫名其妙地受到好处然后加速（除了 SIMDbushi），那我们现在就来分析一下这一次为什么 PT 并行化又让我们的 hash 过程加速了。

在 hash 阶段各进程独立计算 MD5 哈希没有进程间通信开销，工作负载均匀分布。

```
if (global_curr_num > 1000000)
{
    auto start_hash = system_clock::now();
    bit32 state[4];
    for (string pw : q.guesses)  // 各进程独立计算自己的猜测
    {
        if (test_set.find(pw) != test_set.end()) {
            cracked += 1;
        }
        MD5Hash(pw, state);  // 真正的并行计算
    }
    auto end_hash = system_clock::now();
}
```

而相比之下的 guessing 过程中就像我们之前分析的那样，非常容易出现进程之间沟通开销过大的问题，从而使得并行化的优势完全体现不出来。

5 hash 与 guess 并行计算

在这个阶段中我们的主要思想如下

```
串行模式: 生成 1 - 哈希 1 - 生成 2 - 哈希 2 - ...
重叠模式: 生成 1 - 生成 2 - 生成 3 - ...
```

```
-   -   -
哈希 1 - 哈希 2 - 哈希 3 - ...
```

5.1 算法实现

在这个阶段中因为只涉及到两个阶段的调度问题，所以我们只需要更改 correctness_guess.cpp 即可。

```
// 启动独立的哈希计算线程
thread hash_thread(hash_worker_thread, ref(test_set), ref(time_hash));

// 主线程：生成密码猜测
if (!q.priority.empty()) {
    q.PopNextBatch(BATCH_SIZE);
}

// 同时，将猜测转交给哈希线程处理（重叠的关键）
if (!q.guesses.empty()) {
    lock_guard<mutex> lock(guesses_mutex);
    pending_hash_guesses.insert(pending_hash_guesses.end(),
                                q.guesses.begin(), q.guesses.end());
}
```

在这里我们将 hash 处理的函数进行封装，打包成一个独立的函数。hash 过程将在这里进行，同时我们在这里引入了一个休眠的语句 $this\_thread :: sleep\_for(chrono :: milliseconds(1))$; 以保证两个过程之间的协调性。(事实证明这一过程是非常有必要的，我们在前面的探究就可以看出一个非常显著的问题：hash 过程的工作负载是大于 guess 的，这就是为什么我们在实施各种各样的优化方法的时候很容易优化到 hash 头上了)

```
// 哈希计算线程函数
void hash_worker_thread(const unordered_set<string>& test_set, double& time_hash) {
    auto start_hash = system_clock::now();

while (!hash_thread_should_exit) {
    vector<string> local_guesses;

    // 从待处理队列中取出猜测进行哈希
    {
        lock_guard<mutex> lock(guesses_mutex);
        if (!pending_hash_guesses.empty()) {
            local_guesses = move(pending_hash_guesses);
            pending_hash_guesses.clear();
```

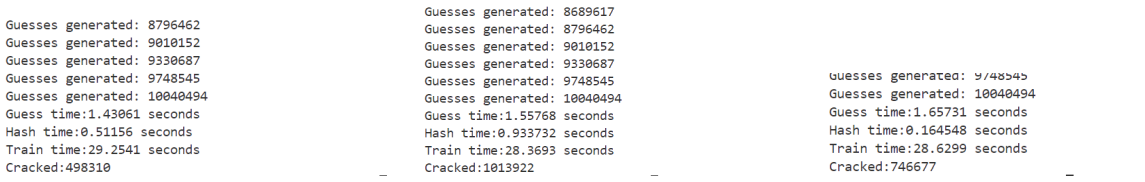
```
            }
        }

        // 进行 MD5 哈希计算
        if (!local_guesses.empty()) {
            bit32 state[4];
            for (const string& pw : local_guesses) {
                if (test_set.find(pw) != test_set.end()) {
                    total_cracked++;
                }
                MD5Hash(pw, state);
            }
        } else {
            // 没有工作时短暂休眠
            this_thread::sleep_for(chrono::milliseconds(1));
        }
    }

    auto end_hash = system_clock::now();
    auto duration = duration_cast<microseconds>(end_hash - start_hash);
    time_hash += double(duration.count()) * microseconds::period::num / microseconds::period::den;
    }
```

因为在实际应用中，我们需要人为地制造延迟使得并行部件之间的通信得到统一，在下面这个地方我们便人为地制造了一个“气泡”使得他们之间地时间得到一个统一与等待。

```
// 主线程生成完猜测后
lock_guard<mutex> lock(guesses_mutex);  // 互斥锁等待
pending_hash_guesses.insert(...);       // 数据传递延迟

// 哈希线程需要等待
if (!pending_hash_guesses.empty()) {    // 检查延迟
    local_guesses = move(pending_hash_guesses);
}
```

5.2 结果展示

在这里我们观察到了一个非常有意思的在之前的实验中从未观察到过的现象：随着我们编译优化的不断深入，我们口令猜测的正确性也在进一步增加。解释这个现象也很简单–那就是我们的文件结束条件出现了问题。



(a) hash-guess 并行化不优化 (b) hash-guess 并行化 o1 (c) hash-guess 并行化 o2

图 5.3: hash-guess 并行化结果

```
int should_exit = 0;
    if (rank == 0 && global_curr_num >= 100000)
    {
        cout << "Guesses generated: " << history + global_curr_num << endl;

        if (history + global_curr_num > 10000000)
        {
            should_exit = 1;
        }
    }
```

在我们使用 hash-guess 并行化的代码中，我们的终止条件选择了生成一千万个猜测的时间点作为结束标志，而不管 hash 过程是否结束，所以我们便可能出现以下情况：而我们知道 hash 所使用的时间又是明显长于 guess 的。而我们可以看出来在编译优化的情况下，guess 的时间相比于之前是变慢了的。这就导致了我们在不同优化情况下得到的 cracked 总数不同，其实是因为我们进行猜测的总次数不同所造成的。

```
时间轴:
主线程:    生成 1000 万猜测 ----------------------------- 退出
哈希线程:  处理 100 万 --- 处理 200 万 --- 处理 300 万 ----- 被强制停止
        |                                  |
    哈希速度慢                          还有 700 万未处理

时间轴:
主线程:    生成 1000 万猜测 ------------------------- 退出
哈希线程:  处理 500 万 --- 处理 800 万 --- 处理 950 万 --- 被停止
        |                              |
    哈希速度快                      只剩 50 万未处理

```

6 为什么我们实施的大多数有效的加速都会使得 hash 过程加速

6.1 Hash 过程是真正独立并行的

hash 的独立并行性在以下几个方面可以得到体现：(1) 无数据依赖：各个计算单元之间没有数据依赖关系。(2) 无同步需求：不需要等待其他计算单元完成。(3) 无通信开销：不需要与其他进程或线程交换数据。(4) 线性加速：理论上可以获得接近进程数倍的加速比。

```
void hash_worker_thread(const unordered_set<string>& test_set, double& time_hash) {
while (!hash_thread_should_exit) {
    // 从队列取出密码
    local_guesses = move(pending_hash_guesses);

    // 独立进行 MD5 计算，无进程间通信
    for (const string& pw : local_guesses) {
        if (test_set.find(pw) != test_set.end()) {
            total_cracked++;
            }
        MD5Hash(pw, state);  // 纯粹的 CPU 计算
        }
    }
}
```

6.1.1 内存访问独立

进程 0 和进程 1 只访问自己的内存区域，并且没有共享内存竞争。

```
// 每个进程有自己独立的数据结构
unordered_set<string> test_set;          // 各进程的本地副本
vector<string> local_guesses;            // 各进程的本地密码
atomic<int> total_cracked(0);           // 各进程的本地计数
bit32 state[4];                          // 各进程的本地 MD5 状态
```

6.1.2 计算逻辑独立

```
// MD5Hash 函数内部
void MD5Hash(const string& input, bit32 state[4]) {
// 纯函数：输入相同，输出确定
// 无全局状态依赖
// 无副作用
for (int i = 0; i < 64; i++) {
    // 位运算、循环移位等基本 CPU 指令
```

```
        // 完全独立于其他进程的计算
        }
    }
```

6.2 不同优化对两个过程的影响差异

在编译优化（-O2/-O3）的情况下，对 Hash 过程的影响显著：4 个进程 = 4 个独立的哈希线程，并且每个线程处理不同的密码，完全并行，最终可以实现加速比接近进程数。

```
// 这些操作受益于编译优化
for (const string& pw : local_guesses) {
    // 字符串查找优化：哈希表访问优化
    if (test_set.find(pw) != test_set.end()) {
        total_cracked++;  // 原子操作优化
    }
    MD5Hash(pw, state);  // 算法内部循环展开、向量化
}
```

但是对 Guess 过程的影响有限。更多进程意味着更多同步点，通信开销随进程数增加，甚至出现了我们结果中的负加速的情况。

```
// MPI 通信时间不受编译优化影响
MPI_Allreduce(...);  // 网络延迟依然存在
MPI_Bcast(...);      // 同步等待依然存在
```

6.3 算法复杂度的差异

对于 hash 算法来说：时间复杂度为 O(n)，并行程度为完美并行，并行之后的复杂度为 O(n/p)，其中 p 为进程数。在最后的通信复杂度：O(1)，因为仅在最后收集结果。对于 guess 来说：时间复杂度为 O(PT 数量 × 每个 PT 的处理时间)，其并行度受限于 PT 间依赖和同步而且通信成本也很复杂，复杂度为 O(循环次数 × 进程数)。

7 将结束标志改为 hash 产生 10000000 个结果

7.1 新增哈希计数器

```
atomic<int> total_hashed(0);  // 统计实际哈希处理的密码数量
```

7.2 在哈希线程中统计处理数量

```
for (const string& pw : local_guesses) {
if (test_set.find(pw) != test_set.end()) {
    total_cracked++;
}
MD5Hash(pw, state);
total_hashed++;  // 每处理一个密码就递增
}
```

7.3 收集全局哈希处理数量

```
// 收集所有进程的哈希处理数量
int local_hashed_count = total_hashed.load();
int global_hashed_count;
MPI_Allreduce(&local_hashed_count, &global_hashed_count, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

7.4 修改终止条件

```
// 终止条件：哈希处理达到 1000 万
if (global_hashed_count >= 10000000) {
    cout << "Reached 10,000,000 hashed passwords. Terminating..." << endl;
    should_exit = 1;
}
```

7.5 结果展示

```
Generated: 48378903 passwords, Hashed: 8500491 password
Generated: 48476226 passwords, Hashed: 8519620 password
Reached 10,000,000 hashed passwords. Terminating...
Waiting for remaining passwords to be processed...
=== Final Results ===
Total passwords generated: 52487847
Total passwords hashed: 22122526
Total passwords cracked: 4705741
Guess time: 2.90776 seconds
Hash time: 3.58335 seconds
Train time: 29.276 seconds
Crack rate: 21.2713%
```

(a) 改变终止条件不优化

```
Generated: 38557937 passwords, Hashed: 8061019 passwords
Generated: 38879942 passwords, Hashed: 8079353 passwords
Generated: 39304580 passwords, Hashed: 8099216 password
Reached 10,000,000 hashed passwords. Terminating...
Waiting for remaining passwords to be processed...
=== Final Results ===
Total passwords generated: 42678797
Total passwords hashed: 18831231
Total passwords cracked: 4117913
Guess time: 2.43824 seconds
Hash time: 3.07508 seconds
Train time: 29.5975 seconds
Crack rate: 21.8675%
```

(b) 改变终止条件 o1

```
Generated: 38557937 passwords, Hashed: 8083021 passwords
Reached 10,000,000 hashed passwords. Terminating...
Waiting for remaining passwords to be processed...
=== Final Results ===
Total passwords generated: 42678797
Total passwords hashed: 29670465
Total passwords cracked: 4899543
Guess time: 6.90734 seconds
Hash time: 3.63466 seconds
Train time: 28.4426 seconds
Crack rate: 16.5132%
```

(c) 改变终止条件 o2

图 7.4: 改变终止条件的时间

我们在这里可以看出来，即使我们最终采用了以产生 10000000 个 hash 结果作为我们的终止条件，但是因为信息的传递的不及时性，我们还是出现了统计上的缺失和不及时。现在我们来分析一下原因。

7.5.1 MPI 与多线程混合环境的固有挑战

这种混合架构中，虽然 total_hashed 是原子变量，但 MPI 通信与线程更新之间没有严格同步，导致读取时的值可能处于不同状态。

```
// 主线程中使用 MPI 通信
MPI_Allreduce(&local_hashed_count, &current_global_hashed, 1,
MPI_INT, MPI_SUM, MPI_COMM_WORLD);

// 同时，哈希线程独立运行
void hash_worker_thread() {
    // ... 处理密码并更新 total_hashed ...
    total_hashed++;  // 原子操作
}
```

7.5.2 时序依赖性问题

这 500 毫秒的等待不足以确保所有进程的哈希线程完成处理，尤其是当有大量密码在队列中等待处理时。

```
if (true_global_hashed >= 10000000) {
should_exit = 1;
}

// 广播退出信号
MPI_Bcast(&should_exit, 1, MPI_INT, 0, MPI_COMM_WORLD);

// 但此时各进程的哈希线程仍在处理数据
this_thread::sleep_for(chrono::milliseconds(500));
hash_thread_should_exit = true;
```

7.5.3 数据一致性窗口

分布式系统中存在”一致性窗口”的概念：读取计数器时，系统需要一个”快照”，但在获取快照的过程中，计数器可能继续变化。所以最终统计时，各进程可能处于不同状态

7.5.4 通信模式的影响

这些通信操作是在不同时间点、不同系统状态下执行的，本质上就是对”移动目标”进行测量。

```
// 循环中使用
MPI_Allreduce(&local_hashed_count, &current_global_hashed,
1, MPI_INT, MPI_SUM,MPI_COMM_WORLD);
```

```
4
5   // 最终统计时使用不同的通信模式
6   MPI_Reduce(&local_cracked, &total_cracked_final, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

8 代码链接

Github 链接：https://github.com/Mercycoffee12138/guess