南開大學
Nankai University

计算机学院

并行程序设计报告

多进程编程实验

姓名：高景珩

学号：2310648

专业：计算机科学与技术

2025 年 6 月 8 日

目录

1 实验说明

这次实验基于多线程的基础上继续进行多进程 MPI 的编程实验，主要针对 Generate() 函数的一个 PT 生成多个口令，还有之后尝试进行改进 popNext() 函数。

2 基础要求——Generate() 函数的并行与加速

2.1 改进思路

2.1.1 guessing.cpp 文件

为实现 PCFG 算法的并行化，本研究基于串行代码设计并实现了基于 MPI 的并行优化方案，重点针对 PriorityQueue::Generate 函数中生成猜测的循环部分进行改进。在串行版本中，Generate 函数通过遍历 PT 对象中最后一个 segment 的全部 value（ordered_values）生成猜测，这一过程是单线程执行，计算开销随 value 数量增加而显著增长。并行版本通过 MPI 分布式计算框架，将猜测生成任务分配到多个进程（rank），每个进程仅处理 value 数组中按 rank 索引分配的子集（通过 for (int i = rank; i < total_vals; i += size) 实现），从而实现任务的负载均衡。进一步地，为减少通信开销，采用序列化技术将各进程的 local_guesses 整合为单一的 char 缓冲区，使用'\0' 分隔字符串，并通过 MPI_Gather 和 MPI_Gatherv 收集各进程的猜测数量和数据，仅在 root 进程（rank 0）进行反序列化和结果合并。这种设计有效降低了通信频率和数据传输量，同时确保了所有猜测按序收集到全局 guesses 容器中。此外，MPI_Barrier 用于同步各进程，确保一致性。改进的关键思路在于：通过任务分解实现计算并行化，通过高效序列化和通信优化降低 MPI 开销，从而在多核或分布式环境下显著提升算法性能。

Algorithm 1 PriorityQueue::Generate 函数流程

Input: 猜测模板 `pt`（包含 `content`、`curr_indices` 和 `max_indices`）

Output: 本地猜测列表 `local_guesses`,根进程更新全局 `guesses` 和 `total_guesses`

1: 获取当前进程编号 `rank` 和进程总数 `size`

2: 调用 `CalProb(pt)` 计算概率

3: 初始化本地猜测列表 `local_guesses`

4: **if** `pt.content.size() == 1` **then**

根据 pt.content[0].type 选择对应段 a(m.letters、m.digits 或 m.symbols)
total_vals ← pt.max_indices[0]
for i = rank to total_vals - 1 step size **do**
将 a->ordered_values[i] 添加到 local_guesses
end for
else
初始化空字符串 prefix
for seg_idx = 0 to pt.content.size() - 2 **do**
idx ← pt.curr_indices[seg_idx]
根据 pt.content[seg_idx].type 选择段，追加 ordered_values[idx] 到 prefix
end for
last ← pt.content.size() - 1
根据 pt.content[last].type 选择段 a
total_vals ← pt.max_indices[last]
for i = rank to total_vals - 1 step size **do**
将 prefix + a->ordered_values[i] 添加到 local_guesses
end for
end if
local_count ← local_guesses 的大小
local_bytes ← local_guesses 中所有字符串长度加终止符之和
使用 MPI_Gather 收集所有进程的 local_count 到 all_counts
使用 MPI_Gather 收集所有进程的 local_bytes 到 all_bytes
if rank == 0 **then**
计算 displs（各进程数据偏移量）及 total_bytes（总字节数）
last_local_counts ← all_counts
end if
将 local_guesses 序列化为字符串 flat，每个猜测后添加终止符
使用 MPI_Gatherv 将 flat 收集到根进程的 recv_buf
if rank == 0 **then**

初始化 offset ← 0
while offset < recv_buf.size() **do**
从 recv_buf[offset] 提取猜测字符串 guess
将 guess 添加到 guesses
offset ← offset + guess.size() + 1
end while
total_guesses ← total_guesses + guesses.size()
end if
MPI_Barrier 同步进程

但是在运行时,出现了报错“Attempting to use an MPI routine before initializing MPICH”，为了解决这个问题，对主函数（测试时选择 correctness_guess.cpp）进行一定的修改。

2.1.2 correctness_guess.cpp(main.cpp) 文件

为了初始化,一开始选择在 main() 函数开始 MPI_Init,结束时使用 MPI_Finalize,但是运行发现，这样操作会使得函数在训练等阶段同时使用多进程，在使用-np 参数为 4 时，模型就对应被训练 4 次，显然不符合我们的预期，所以就对主函数文件进行优化，思路大概如下:

首先，在 main 函数中，仅 rank 0 进程执行模型训练（q.m.train）和初始优先队列构建（q.init），随后通过 BroadcastModel 和 BroadcastPT 函数将模型数据和当前 PT 对象广播至所有进程，确保各进程拥有一致的输入数据。其次，猜测生成任务通过并行化的 Generate 函数分配到各进程，进程按 rank 和 size 分担 ordered_values 的子集生成局部猜测（local_guesses），并通过 MPI_Gather 和 MPI_Gatherv 高效收集至 rank 0。各进程并行生成猜测和计算哈希，通过 MPI_Reduce 汇总结果；使用 MPI_Barrier 和 MPI_Bcast 确保进程同步和数据一致；全局猜测计数和终止条件由根进程管理，优化内存使用和计算效率。

具体思路如下:

Algorithm 2 MPI 优化的主函数流程

Input: 训练数据文件，测试数据文件，猜测上限 max_total_guesses

Output: 猜测时间、哈希时间、检查时间、训练时间、破解数量

初始化 MPI 环境，获取 world_rank 和 world_size
if world_rank == 0 **then**
    使用 q.m.train 和 q.m.order 训练模型
end if
MPI_Barrier 同步进程
调用 BroadcastModel 广播模型数据
各进程加载测试数据到 test_set（最多 1000000 条）
if world_rank == 0 **then**
    初始化优先级队列 q.init()
end if
初始化全局猜测计数器 global_total_generated 和破解总数 total_cracked
记录全局开始时间 t_global_start
while q.priority 不为空 **do**
    if world_rank == 0 **then**
        从 q.priority 获取 current_pt
        删除 q.priority 首元素
    end if
    调用 BroadcastPT 广播 current_pt
    各进程调用 q.Generate(current_pt) 生成猜测
    本地计算 MD5 哈希，记录 local_hash_time
    本地检查猜测，记录 local_cracked 和 local_check_time
    使用 MPI_Reduce 汇总 local_generated、local_cracked、local_hash_time 和 local_check_time
    if world_rank == 0 **then**
        更新 global_total_generated 和 total_cracked
        if global_total_generated $\geq$ 500000 **then**
            打印当前猜测总数
        end if
        if global_total_generated $\geq$ max_total_guesses **then**
            记录全局结束时间，计算 time_guess

30: 打印时间和破解数量
31: 广播 terminate 信号
32: **end if**
33: **end if**
34: 广播 terminate，若为真则退出循环
35: 清空 q.guesses
36: MPI_Barrier 同步进程
37: **end while**
38: MPI_Finalize

2.2 性能测试

对 MPI 并行化后的代码版本进行性能测试，以串行为 baseline 得到以下数据表：

guess_time	串行时间 (-O0 编译优化)	MPI 并行时间 (-O0 编译优化，8 进程)	串行时间 (-O2 编译优化)	MPI 并行时间 (-O2 编译优化，8 进程)
1	7.77224	4.11829	0.594009	0.747247
2	7.77631	4.83923	0.611865	0.703292
3	7.75978	4.86196	0.601064	0.72839
4	7.75989	4.5452	0.614346	0.606939
5	7.80014	4.43553	0.57522	0.702186
平均时间	7.773672	4.560042	0.5993008	0.6976108

可以看到在 O0 编译优化下有一定的加速提升，但是在 O2 编译下性能略微降低，O0 优化加速比大约为 1.705。

3 进阶改进——数据结构

沿着上次的思路，我们不使用 vector 向量来存储 priority，而使用优先队列来维护，经过验证，程序的相关时间复杂度从 O(N) 的插入复杂度降低到 O(log N)。

除此之外，为了进一步优化程序，我把 hash 的过程也放入各个子进程中，所以 hash_time 也进一步下降。

进过简单测试，可以得到如下两个在-O0 和-O2 编译优化下的测试结果输出图：

```
--- Final Report ---
Total guesses generated: 10035780
Total passwords cracked: 355177
--------------------
Train time:          97.7856 s
Total guess time:    2.6599 s
  - Hashing time:      1.2152 s
  - Checking time:     1.0469 s
--------------------
```

图 3.1: -O0 编译优化的结果

```
--- Final Report ---
Total guesses generated: 10035780
Total passwords cracked: 355177
--------------------
Train time:          24.0496 s
Total guess time:    1.1735 s
  - Hashing time:      0.4041 s
  - Checking time:     0.5293 s
--------------------
```

图 3.2: -O2 编译优化的结果

计算可得加速比范围在 2.92 左右

4 进阶改进——多 PT 处理

与上次不同的是，因为这次的 main 函数改动较多，无法直接通过修改 PopNext 函数来进行直接的一步替换多个 PT 的处理，但是思路大致是相同的，大致修改如下：

主进程首先对训练集进行模型训练和排序，然后通过 BroadcastModel 与 BroadcastTestSet 将模型和测试集广播到所有进程。主循环中，主进程每次从优先队列中提取多个高优先级的 PT 结构，并广播到所有进程。各进程并行调用 Generate 函数生成多个口令猜测，并分别执行哈希与测试集命中判断操作，结果通过 MPI Reduce 聚合至主进程，进行统计输出与终止条件判断。该结构通过批量 PT 分发和分布式口令生成提升整体吞吐率，同时保留了模型驱动的优先队列调度逻辑。

经过测试，可以得到和上次类似的结果。

```
--- Final Report ---
Total generated: 10038830
Guess time: 0.547978 seconds
Hash time: 1.9424 seconds
Train time: 92.7099 seconds
```

图 4.3: -O0 多 PT 结果

```
--- Final Report ---
Total generated: 10038830
Guess time: 0.302113 seconds
Hash time: 0.268574 seconds
Train time: 25.4674 seconds
```

图 4.4: -O2 多 PT 结果

计算可得，并行加速比达到了 14.1，运行效率大幅提升。

5 总结

本次实验通过 MPI 并行化框架对 PCFG 算法的 `Generate` 函数进行了优化，实现了任务分解、负载均衡和高效通信，显著提升了程序性能。基础并行化方案通过进程分配和序列化技术，在 -O0 编译优化下获得约 1.705 的加速比。进阶优化通过替换 `vector` 为 `priority_queue` 和并行化哈希计算，进一步将加速比提升至 2.92。多 PT 处理方案通过批量分发 PT 和分布式猜测生成，将加速比提升至 14.1，展现了 MPI 在分布式计算中的强大潜力。

然而，实验也发现，在 -O2 编译优化下，MPI 并行版本的性能略低于串行版本，可能是由于编译优化对串行代码的改进效果更显著，而 MPI 的通信开销在高优化级别下变得相对突出。未来可进一步优化通信机制，如减少 `MPI_Barrier` 的使用或采用异步通信，以进一步提升性能。

6 心得

6.1 实验心得

通过本次实验，我深入理解了 MPI 并行编程的原理和实现方法。任务分解、负载均衡和通信优化是并行计算的核心，MPI 提供的 `MPI_Gather`、`MPI_Gatherv`、`MPI_Reduce` 和 `MPI_Barrier` 等接口为分布式计算提供了强大支持。在优化 `Generate` 函数时，我学会了如何通过索引分配实现任务均分，并通过序列化技术降低通信开销。数据结构的优化（如 `priority_queue`）让我认识到算法效率对并行程序的重要性，而多 PT 处理的成功实现进一步加深了我对批量任务分发的理解。

实验中遇到的“MPICH 未初始化”问题让我认识到 MPI 环境配置的重要性，通过调整 `main` 函数逻辑，成功避免了模型重复训练的问题。此外，性能测试结果让我意识到编译优化对并行程序的影响复杂，需要综合考虑计算和通信的平衡。总体而言，本次实验让我从理论到实践全面提升了对并行程序设计的理解。

6.2 理论分析

6.2.1 MPI 并行受编译优化的影响

在高性能计算中，编译器优化对 MPI 并行程序的性能影响呈现出复杂的非线性特征。实验结果显示，当关闭编译优化（`-O0`）时，MPI 并行版本通过将串行代码中低效的循环计算分摊至多核执行，实现了约 1.7 倍的加速比，此时串行代码的循环迭代、函数调用等操作存在显著优化空间，并行化带来的计算分摊效应占主导地位。然而，当启用中等优化级别（`-O2`）后，串行程序的执行时间从约 7.8 秒骤降至 0.6 秒，而 MPI 并行版本仅降至 0.7 秒，并行加速效果被显著削弱。

这一现象的本质原因在于编译优化与并行开销的博弈：一方面，`-O2` 通过循环展开、函数内联、常量传播及自动向量化（如将标量运算转换为 AVX 指令集的 SIMD 操作）等技术，将单核计算效率推向硬件极限——例如，连续访问数组的循环被优化为缓存友好的顺序模式，分支预测错误率从 30% 降至 5% 以下，L1 缓存命中率提升至 90% 以上，使得串行代码的计算时间压缩近 13 倍；另一方面，MPI 并行引入的固定通信开销（如数据序列化、MPI_Gatherv 网络传输、MPI_Barrier 同步）并未随编译优化减少，当计算时间从秒级降至毫秒级时，这些微秒级的开销占比从不足 5% 飙升至 50% 以上。具体而言，分布式场景下的跨步数据访问（如

各进程按 rank 索引拆分 ordered_values 数组）导致无法利用 CPU 的自动向量化能力，而 MPI 库函数的黑箱特性（编译器无法对其内部逻辑进行内联或循环展开）进一步使得通信代码维持在低优化水平。这种计算效率与通信开销的失衡，最终导致在高编译优化级别下，并行化的收益被固定开销抵消，甚至出现性能倒退。

此现象揭示了并行程序设计的核心挑战：当计算效率接近硬件理论峰值时，任何非计算性开销（即使是微秒级的通信延迟）都可能成为性能瓶颈，需在算法设计阶段通过粗粒度任务划分、异步通信重叠计算等策略重构计算-通信平衡。

6.2.2 并行效率的理论模型与通信–计算平衡

从 Amdahl 定律的理论框架出发，程序的加速比受限于串行部分的比例，即 $S \leq \frac{1}{F+(1-F)/P}$（其中 F 为串行代码占比，P 为进程数）。在 PCFG 算法中，模型训练阶段的统计计算、优先队列的全局排序逻辑（如基于概率的 PT 选择）以及根进程的结果汇总操作，均属于不可并行的串行瓶颈。即使将 Generate 函数的并行度提升至极限，这些串行模块仍会导致加速比存在理论上限——例如，若串行部分占比 $F = 5\%$，则无论进程数多少，加速比无法超过 20 倍。通信–计算比（CCR）的失衡是实验中性能倒退的核心机制。该指标定义为单次通信开销与计算任务耗时的比值，当 $\mathrm{CCR} \ll 1$ 时，并行效率趋近于理想线性加速；反之，当 $\mathrm{CCR} \geq 1$ 时，通信延迟将主导总耗时。在 $-$O2 优化下，单个进程处理的 value 子集计算耗时从毫秒级降至微秒级（如每个进程处理 1000 个元素的时间从 1ms 降至 0.1ms），而每次 MPI_Gatherv 的通信延迟（含序列化、网络传输、同步）约为 100 s，导致 $\mathrm{CCR} \approx 1$。此时，并行程序的实际执行时间可近似为 $T_{\text{parallel}} = T_{\text{compute}} + T_{\text{comm}}$，而串行程序经优化后 $T_{\text{serial}} \approx T_{\text{compute}}$，从而出现并行性能劣于串行的现象。这种 “计算被通信吞噬” 的效应，在任务粒度过细或编译优化显著提升计算效率时尤为明显。

6.2.3 三层并行范式的协同提升

现代高性能计算通过混合并行模型（Hybrid Parallelism）实现多层级优化，其核心在于将不同粒度的并行需求分配至最合适的抽象层。在 PCFG 算法的优化中，可构建以下三层协同框架：

1. 分布式并行（MPI）：负责跨节点的粗粒度任务分发，例如将 PT 集合按优先级分片至不同节点，每个节点仅处理全局 PT 队列的子集，减少跨节点的数据传

输量。通过 MPI_Bcast 同步模型参数（如字符段概率表），利用 MPI_Reduce 聚合各节点的破解结果，避免根节点成为通信瓶颈。

2. 共享内存并行（OpenMP）：在单节点内，对 Generate 函数中的循环进行多线程并行化。例如，将每个进程（MPI rank）的 local_guesses 生成任务进一步拆解为线程级并行，利用 CPU 多核并行填充有序值子集，同时通过线程私有缓存减少内存竞争。此层优化可将计算效率提升 2–4 倍（取决于 CPU 核心数），并降低单位计算的通信开销占比。

3. 指令级并行（SIMD）：通过编译器自动向量化（如 GCC 的 −ftree-vectorize 选项）或手动编写 AVX 指令，将字符拼接、哈希计算等操作转换为向量指令。例如，将 16 个字符的 MD5 哈希计算合并为单条 AVX2 指令，使计算速度提升 4–8 倍。此层优化对串行和并行代码均有效，但在并行环境中可通过减少单线程计算时间间接降低通信开销比例。

三层范式的协同效应体现在：分布式层减少跨节点通信量，共享内存层提升单节点计算效率，指令级层压缩单线程计算耗时，最终形成“节点间低通信、节点内高计算、计算内高吞吐”的优化路径。实验表明，当三层优化均被启用时，CCR 可从 1 降至 0.1 以下，使并行加速比突破 Amdahl 定律的理论限制（通过减少串行部分占比 F），接近理想线性加速。这种多层次优化策略，本质上是通过体系结构感知的任务分解，将计算负载精准匹配至硬件的多层次并行资源（集群网络、多核缓存、向量寄存器），从而实现性能的全方位提升。

7 代码仓库

这是项目的代码地址：github 仓库地址。