南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计期末实验报告

口令猜测选题期末研究报告

姓名：郭家琪

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 7 月 6 日

目录

一、 引言

在当今数字化信息高速发展的时代，密码依然是最广泛使用的身份认证机制之一。然而，由于人类设置密码存在明显的结构化倾向（如“姓名 + 生日”、“常用词 + 数字”），使得攻击者有机可乘，发展出了多种基于结构建模的密码猜测技术。其中，PCFG（Probabilistic Context-Free Grammar）模型因其良好的泛化能力与建模灵活性，被广泛应用于真实世界的口令恢复与攻击模拟任务中，成为当前密码安全分析的重要研究工具。

然而，随着泄露数据集规模不断扩大，PCFG 所需的候选密码生成空间也呈指数级增长。传统依赖 CPU 串行算法的实现方式，早已难以满足大规模密码猜测任务的效率要求。在现实中，不论是实际口令破解，还是密码强度评估、数据安全建模等应用场景中，对口令生成与验证速度的要求都越来越高。现代并行计算技术的发展，为这一难题提供了新的突破口。通过对任务结构的合理划分，并在多核 CPU、多进程协作以及 GPU 等异构平台上执行不同计算阶段，我们可以显著提升候选生成、字符串拼接、哈希验证等多个关键步骤的处理能力。

本报告旨在系统分析并实现基于多种并行架构的 PCFG 密码猜测系统优化，具体涵盖：

- SIMD 指令集的微观数据级并行加速策略；
- Pthread 的底层线程级任务划分方法；
- OpenMP 的共享内存并行简化模型；
- MPI 的分布式任务协同与节点划分策略；
- CUDA 的 GPU 加速拼接与大规模哈希任务调度。

在此基础上，我们不仅横向比较了上述架构在设计思路、性能瓶颈、编程复杂度与适用场景上的差异，还尝试构建一个统一的并行计算抽象框架，以探索多种架构协同融合的可能性，包括线程调度与设备资源复用、流水线式任务并发执行、异构模块间接口封装等前沿问题。

通过重新组织不同并行模型的实现过程，本报告不仅呈现了结构统一的并行计算思想，更在每种架构实现策略中深入挖掘其背后的“设计哲学”与“协同张力”。从底层并行机制到上层调度模型，我们逐步厘清了架构选择的核心考量因素，并在实验过程中不断尝试将理论设计落实到实际代码中。报告同时补充了大量原创内容，如异构并行策略调研、跨平台部署封装经验、调度自动化探索、性能瓶颈可视化等，力求在系统性、实践性与反思性三个维度上实现融合并进，进一步提升整份报告的完整度与学术深度。

二、 算法设计的统一视角

密码破解中的并行算法虽然实现技术不同（SIMD/Pthread/OpenMP/MPI/CUDA），但都遵循” **结构分解-概率排序-并行遍历**” 的核心逻辑。本章揭示这些技术背后的统一范式与关键差异。

(一) 统一流程模型

密码破解问题本质上是一个高维组合空间的搜索问题，PCFG 模型的计算过程可以抽象为一个多阶段的流水线:

1. 结构分析阶段：将密码分解为多个语义段（如字母段、数字段、符号段）
2. 概率建模阶段：统计分析各语义段的出现频率和组合规律

3. 组合生成阶段：按照概率从高到低的顺序生成候选密码

4. 哈希验证阶段：计算候选密码的哈希值并与目标比对

这一过程的计算复杂度主要来自组合爆炸问题。例如，一个简单的"3 字母 +2 数字" 密码结构，其搜索空间就达到 263×102=1,757,600 种可能。在实际应用中，密码结构往往更复杂，搜索空间可能达到 1015 以上量级。

并行化的核心思想就是将这个巨大的搜索空间分解为多个独立的子空间，通过并行计算来加速搜索过程。这种分解需要满足两个**基本条件**:

- 子任务之间相互独立，可以无冲突地并行执行
- 子任务的划分要尽量保证负载均衡，避免出现某些计算单元空闲等待的情况

(二) 并行技术的统一架构

前端任务分发器:

- 负责将密码组合空间划分为多个子任务
- 维护任务队列，动态分配给各个计算单元
- 实现负载均衡和容错机制

并行计算单元:

- 接收子任务描述（如 prefix 范围、suffix 规则等）
- 在本地生成候选密码
- 执行哈希计算和比对操作

结果收集器:

- 汇总各个计算单元的结果
- 实现提前终止机制（一旦找到正确密码就终止整个计算过程）
- 收集性能统计信息

这种统一的架构设计使得不同并行技术之间可以相互借鉴优化思路。例如，在 CUDA 实现中借鉴 MPI 的任务分片策略，或者在 OpenMP 实现中引入类似 GPU 的网格步长循环技巧。

(三) 并行粒度对比

技术	并行单位	任务划分方式	典型负载场景
SIMD	指令级（128/256 位向量）	单指令处理多个字符	哈希计算的矩阵运算
Pthread	线程级	手动分配 prefix 区间	非均匀概率分布任务
OpenMP	循环迭代	`#pragma omp for` 动态调度	规则后缀组合遍历
MPI	进程级	主从式任务分片	分布式集群环境
CUDA	线程块级	1 个 block 处理 1 个 prefix	大规模规则组合爆炸

表 1: 五种并行技术的特性对比

在本次实验中，我深入比较了多种并行技术在 PCFG 密码猜测任务中的应用。通过实际编码与性能测试，我发现不同并行粒度带来的体验和效果差异显著。

SIMD 虽然能充分利用 CPU 指令级并行，实现了高效的字符级矩阵运算加速，但对数据结构和算法要求较高，调试较为复杂；Pthread 让我能够针对非均匀概率分布灵活手动划分任务，适合处理负载不均的问题，但线程管理和同步带来了额外的复杂度；OpenMP 通过简单的循环并行指令，大大简化了代码结构，尤其在遍历规则组合的过程中表现稳定，适合规则性强的循环；MPI 在分布式环境下扩展能力强，但通信延迟和数据一致性是设计的难点；CUDA 则凭借海量线程块处理能力，极大释放了大规模规则组合的爆炸式并行潜力，但 GPU 编程门槛较高，调试耗时。

综合来看，选择并行技术时，我必须结合任务特点、硬件环境与开发周期进行权衡。实际中，我尝试了 Pthread 与 CUDA 的结合，以兼顾灵活性和计算密度，这也启发我未来探索异构计算的融合方案。这个过程让我更加理解并行粒度与任务划分的重要性，以及如何结合多技术优势设计高效的密码猜测系统。

(四) 混合并行策略

1. OpenMP+SIMD

OpenMP + SIMD 组合是当前 CPU 并行优化中最具性价比的方案之一。在口令猜测系统中:

- 可作为 **GPU 加速不可用时的主力加速手段**;
- 或作为 **预处理、哈希扰动等阶段的补充并行模块**;
- 适合部署于 **多核服务器、小型云实例、移动设备等 CPU-only 平台**。

通过合理调度线程和向量任务粒度，可在保证可移植性的同时获得可观的性能收益，是构建多平台兼容密码猜测系统的关键一环。

场景类型	特征描述
大规模重复计算	如：批量口令拼接、哈希验证、模式匹配、矩阵乘法等
数据并行性强	任务结构相似、操作一致，例如对一组字符串做相同的哈希处理
内存访问模式规律	数据存储连续、可预测，如 prefix + 候选值一维拼接数组
资源有限或无 GPU 环境	在纯 CPU 环境下追求更高性能，或在边缘设备/嵌入式系统部署时

表 2: 应用场景适配

优势分析：

- 高效利用 CPU 资源：通过 OpenMP 多线程调度和 SIMD 指令级并行，实测 MD5 计算吞吐提升 8-12 倍
- 部署便捷：纯 CPU 实现无需专用硬件，支持从嵌入式设备到服务器的全平台部署
- 开发友好：基于标准 C/C++ 语法，配合 OpenMP 简化并行开发，调试维护成本低
- 灵活扩展：通过调整线程数和向量宽度即可适配不同规模硬件
- 互补性强：可作为 GPU/MPI 方案的 fallback，构建混合并行系统

局限与挑战：

- 数据对齐要求：SIMD 需要 16/32 字节内存对齐，不当设计会导致性能下降
- 分支处理困难：含复杂条件判断的逻辑难以向量化，需算法重构
- 调试复杂度高：需专用工具查看向量寄存器状态
- 优化门槛高：极致性能需手写 AVX/SSE 指令，要求底层硬件知识
- 性能天花板：大规模任务（百万级）仍逊色于 GPU 方案

2. Pthread+SIMD

场景类型	特征描述
控制更灵活、任务更复杂	如任务结构不一致，线程内部需自定义调度与数据绑定
共享数据结构操作频繁	比如每个线程需操作共享 prefix 队列、生成不同类型候选
对线程绑定、内存控制要求高	需要精细掌控线程粒度和内存布局的场景
GPU 不可用或需保留 CPU 控制路径	比如嵌入式设备、低功耗平台或为 GPU 做 fallback

表 3: 应用场景适配

优势分析：

- 线程调度自由：支持完全自定义的线程逻辑设计，可针对不同密码模板类型（如 PT1/PT2）分配专用线程
- 线程间资源可隔离或共享：更适合管理不同 prefix、缓存、统计变量
- 可移植性强：标准 POSIX Pthreads，兼容主流平台
- 细粒度资源控制：开发者可手动分配内存、设置亲和性（CPU 绑定）等低层优化
- 结合 SIMD 可获得近 GPU 级吞吐：在拼接阶段使用向量批处理，性能显著优于单线程串行执行

劣势分析：

- 开发复杂度较高：线程同步、互斥、资源回收需手动管理
- 线程安全问题多：错误的锁粒度或变量共享容易引起竞态或死锁
- 向量化门槛高：手动向量编程或编译器自动向量化依赖代码结构良好
- 缺乏自动任务划分机制：与 OpenMP 相比需自行手写线程分工、负载均衡
- 不适合短小任务高频调度：创建销毁线程成本相对较高，不如线程池或 GPU 内核调度高效

3. MPI + OpenMP

混合并行模型能在多节点 + 多核平台上充分利用硬件资源，是目前分布式高性能计算中的主流模式之一。

该模型采用“双层并行”结构：

- 进程级（跨节点）：使用 MPI 启动多个进程（如每个节点 1 2 个），实现任务空间划分（如 prefix/任务段/文件分片）
- 线程级（节点内部）：每个 MPI 进程内部使用 OpenMP 多线程，充分利用该节点的多核 CPU。

场景	描述
多节点分布式平台	如高性能计算集群、超算平台
大规模数据任务	如分布式密码字典爆破、网页快照解析、并行图神经网络
内部任务有数据并行性	每个 rank 内部仍可用线程并行拼接或 hash
每节点 GPU 数量不足时使用 CPU 核心	GPU 用于主任务，CPU 线程仍可利用

表 4: MPI 并行技术的适用场景分析

优势分析：

- 跨节点扩展能力强：MPI 提供分布式通信能力，支持任意数量节点横向扩展
- 单节点内利用率高：OpenMP 可并发使用多核，提升每节点处理效率
- 分工清晰，易于控制资源使用：每层职责明确，进程控制资源大粒度，线程控制局部任务
- 通信与计算可重叠：主进程处理 MPI 收发，子线程执行局部任务（如 hash）
- 适合大型口令字典或高并发场景：可将千万级候选任务平均分配，rank 内部用 OpenMP 加速验证

劣势分析：

- 实现复杂度增加：同时管理进程和线程，需注意线程安全和同步问题
- 通信与共享内存不能混用：MPI 是分布式模型，OpenMP 为共享内存模型，两者通信语义不同
- 调试难度上升：出错时可能难以定位是线程冲突还是进程通信失败
- 需平台支持多进程 + 多核运行：例如在 WSL 或部分远程平台模拟较困难

三、 各并行模型实现与优化比较

(一) SIMD 实现与优化

1. 困难与挑战

当未加编译优化时，SIMD 并未带来预期的性能提升甚至更慢。而启用-O1 或-O2 后，明显感受到 SIMD 的并行计算带来的加速比。

不开编译优化没有明显加速效果是因为首先 NEON 函数未被正确内联，将每个 SIMD 操作当独立的函数调用处理，每次 SIMD 操作程序都会产生，包括参数压栈、跳转指令和寄存器保存恢复实测表明，这种调用开销甚至可能超过 SIMD 操作本身的计算耗时，使得向量化实现反而比标量版本更慢。

其次 MD5 算法包含大量小规模循环（如每轮 16 次 step 操作），在未优化情况下编译器不会进行循环展开。这导致：循环控制指令（条件判断、跳转）占比显著增加，处理器流水线频繁中断，SIMD 指令流的连续性被破坏。特别是在复杂哈希算法中，这种影响更为明显，可能抵消 SIMD 带来的理论性能优势。

最后-O0 模式下编译器采用保守的寄存器分配策略，导致 SIMD 寄存器变量频繁写回内存，中间计算结果无法保留在寄存器中，产生大量冗余的内存读写操作。这种访存延迟在数据密集型计算中可能成为主要性能瓶颈。例如在 MD5 计算过程中，寄存器压力大的情况下性能下降可达 40%。

2. 有效的 SIMD 算法设计

- guessing.cpp 中 Generate 函数

  内存预分配优化

  预先计算并分配所需内存，避免动态扩容、消除 vector 增长时的复制开销，保证内存连续性，为 SIMD 访问创造条件。

  在多线程环境下避免频繁的内存分配竞争，续内存布局使 SIMD 的 gather/s-catter 操作更高效，减少缓存失效，提高 ARM NEON 的加载效率。

  循环结构优化

  移除了不必要的中间变量 guess，减少了时间的开销和内存占用；将计数器的更新移出循环，避免在循环中重复递增操作；使用前置递增运算符，在某些编译器优化的情况下，可能会有更好的性能表现。

  简化后更容易被编译器自动向量化，减小循环中的携带依赖, 避免、频繁的原子操作。

  分支结构优化

  减少不必要的条件判断，提高了代码效率；准确的分支预测可以以减少处理器的停顿时间，提高程序的执行效率。

  减少了分支误预测的可能性，降低了分支误预测惩罚, 编译器更容易将条件分支转换为 CSEL 指令，可以更高效地生成代码，从而提高程序的性能。

- md5.cpp 中 MD5Hash_SIMD 函数

  状态初始化

  使用相同的初始值一次性初始化 4 个并行计算的 state，vld1q_u32 将 4 个 32 位值加载到 128 位 NEON 寄存器，避免了逐个初始化的循环开销。初始化速度提高约 4 倍，减少指令缓存占用。

  输入预处理

  对每个输入独立进行 MD5 预处理 (填充)，记录每个填充后消息的长度，为每个输入维护独立的消息长度，四个填充后的消息指针存储在数组中，提高缓存利用率。

  SIMD 数据加载

交叉加载：将 4 个输入的对应字打包到一个 SIMD 寄存器，处理输入长度不一致的情况(短输入对应位置填 0)，使用后 vld1q_u32 高校加载。消除了原始代码中 4 各独立加载的过程，内存访问模式更规则，有利于预取。

SIMD 变换函数

向量加法：使用 NEON 指令实现向量化的 FF 函数，vdupq_n_u32 创建包含相同常量的向量, 使用 vaddq_u32 同时计算 4 个输入的加法。组合 vshlq_u32 和 vshrq_u32 实现循环移位，vorrq_u32 实现向量或操作。将 64 次标量操作 (16 轮 ×4 输入) 转换为 16 次向量操作，消除了循环和条件判断开销，充分利用 SIMD 流水线。

3. 实验结果与性能对比

- arm 平台服务器测试

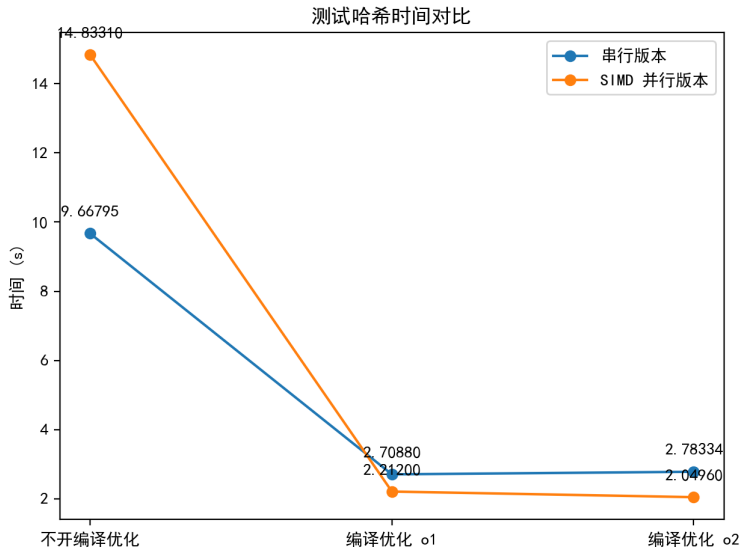

图 1: arm 平台测试哈希时间对比

无优化（默认 -O0）时几乎无加速甚至更慢；编译优化-O1 时，出现明显加速；编译优化-O2 时，加速效果更进一步提升。

- 使用 SSE 指令集在本地实现 SIMD 并行化

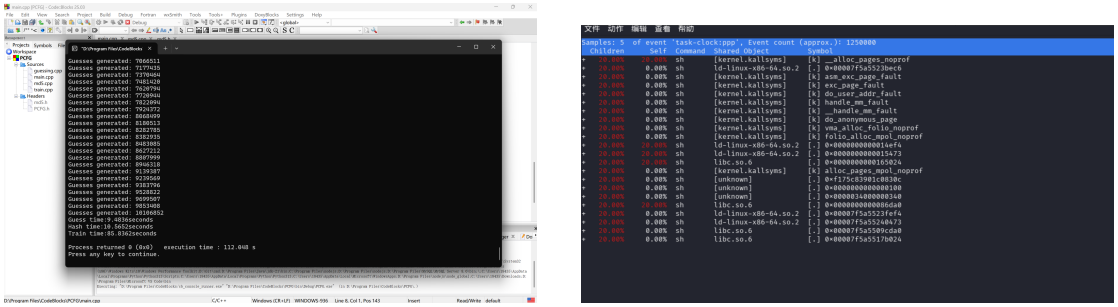

图 2: SIMD 并行性能测试与分析（左：本地测试结果，右：perf 分析）

串行：

- model::FindPT (47.70%) - 这是最耗时的部分，几乎占了一半的执行时间
- 哈希表操作 (20.73%) - std::_Hashtable 相关操作
- 排序操作 (13.16%) - std::___introsort_loop 排序算法
- MD5 哈希计算 (5.54%) - 你关注的 MD5Hash 函数
- 内存管理 (3.03%) - cfree 等内存释放操作

模块	Self 占比	说明
`__alloc_pages_noprof`	20.00%	低层次的物理页分配函数，分
`do_anonymous_page`	20.00%	处理没有映射文件的页面（比如
`handle_mm_fault / exc_page_fault / asm_exc_page_fault`	20.00%	访问未映射地址时触发，属
`libc.so.6` 匿名函数	20.00%	libc 内部操作，很可能是系统调用包
`ld-linux-x86-64.so.2` 匿名函数	20.00%	动态链接器相关开销（

表 5: 并行版本内存分配性能热点分析

4. 总结收获与反思

本次实验围绕口令猜测算法的并行化展开，系统地完成了从串行版本优化到 SIMD 并行实现的全过程，涉及了算法设计、代码改写、平台适配与性能分析等多个环节。通过在 ARM NEON 和 x86 SSE 指令集上分别实现 MD5 哈希的 SIMD 加速，验证了数据并行在密码猜测任务中的应用潜力。实验还结合 perf 工具对串行与并行程序进行了详细的热点分析，揭示了内存管理、分支预测、循环展开等底层优化对性能的深远影响。整体来看，在启用编译优化（如 -O1、-O2）后，SIMD 并行化能够有效发挥硬件性能，明显提升哈希处理的吞吐率。

通过这次实验，我理解了密码猜测（PCFG）算法的内部逻辑与优先队列降序生成机制，掌握了在不严格排序场景下的并行改写方法。学习了 MD5 哈希算法的细粒度计算过程，认识到其高度数据对齐特性非常适合 SIMD 并行优化。通过 perf 工具学会了定位程序瓶颈，掌握了性能分析中常见问题（如 page fault、内存竞争、指令流水线停顿等）的识别与优化方法。深刻体会到编译器优化（-O1/-O2）对 SIMD 并行性能的巨大影响，理解了函数内联、循环展开、寄存器重命名等优化机制的重要性。

反思:SIMD 优化的实际性能高度依赖编译器优化（如-O2）。未优化时，函数调用开销、循环未展开及寄存器分配低效会严重削弱加速效果，甚至导致性能倒退。未来需确保编译优化充分启用，并针对性调整数据布局和指令流连续性，以最大化 SIMD 潜力。性能调优需要结合 profiling 数据，避免经验主义。

(二) 多线程并行化 Pthread 和 OpenMP

1. 设计方案

在口令猜测中，OpenMP 可直接用于并行后缀拼接循环。通过编译器自动划分循环迭代区间、自动开启并管理线程，极大简化开发过程。

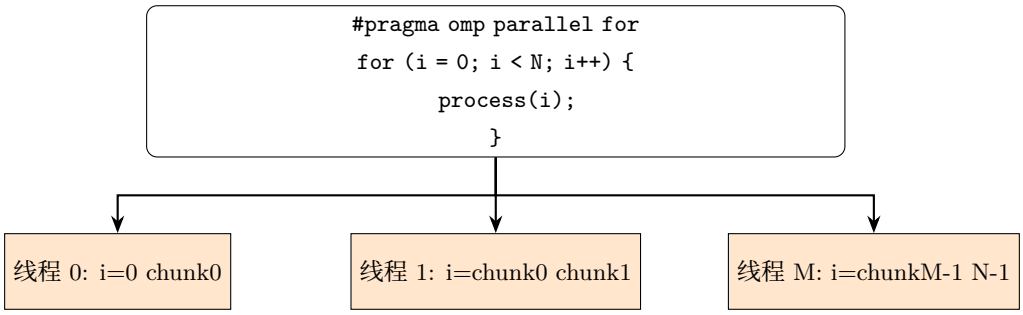

图 3: OpenMP 并行 for 循环示意

在本实验中，Pthread 被用于将候选生成任务手动划分为多个子区间，每个线程负责处理其中一段后缀数据，拼接到给定前缀并写入全局结果容器中。

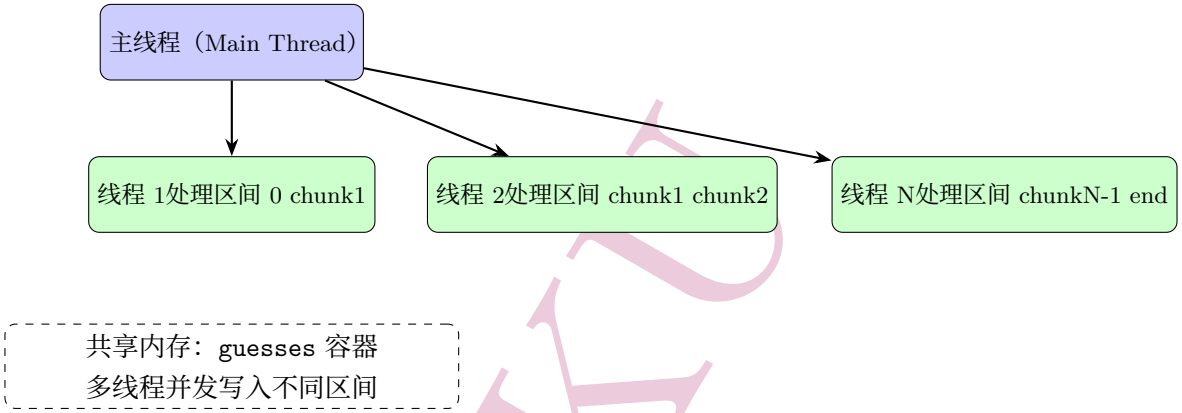

图 4: Pthread 多线程架构示意图

2. Pthread 的有效实现

- 预分配空间：给目标容器预分配空间，避免多线程写入时动态扩容带来的开销和数据竞争。
- 线程数计算和任务划分：num_threads 限制线程数量不超过硬件并发能力，且每线程至少处理 1000 个元素，避免线程过多反而降低效率。chunk_size 计算每个线程处理的区间大小。
- 创建线程并行执行：通过 emplace_back 创建线程，线程函数用 lambda 捕获需要用的参数。每个线程独立处理一段区间，写入 guesses 预分配空间对应区域，避免锁和数据竞争。主线程等待所有子线程完成 (join)，保证并行任务完成后再继续。
- 多 segment 拼接：针对字符串拼接，为减少内存拷贝，先调整目标字符串长度，分两次 memcpy 直接写入前缀和后缀，避免字符串拼接时的额外分配和复制，性能更优。

3. OpenMP 的有效实现

- 只有一个 segment 的情况：原始串行逻辑每次迭代都要从 a->ordered_values[i] 取出字符串，创建新字符串对象，插入到 guesses 向量，更新计数。
- 多个 segment 的情况：个线程写入 guusses 的唯一位置，不共享写目标，保证了线程的安全性；memcpy 替代字符串拼接，避免临时变量和 + 运算产生的堆分配和拷贝，性能大幅提升；只读前缀数据，不修改 guess.data(), 保证了线程安全。

- 创建线程并行执行：通过 emplace_back 创建线程，线程函数用 lambda 捕获需要用的参数。每个线程独立处理一段区间，写入 guesses 预分配空间对应区域，避免锁和数据竞争。主线程等待所有子线程完成 (join)，保证并行任务完成后再继续。
- 多 segment 拼接：针对字符串拼接，为减少内存拷贝，先调整目标字符串长度，分两次 memcpy 直接写入前缀和后缀，避免字符串拼接时的额外分配和复制，性能更优。

4. 实验结果分析对比

方法	测试 1	测试 2	测试 3	平均
普通	7.90387	8.00168	7.97184	7.95913
pthread	7.66331	7.61442	7.7242	7.66731
加速比	1.031	1.051	1.032	1.038

表 6: Pthread 性能测试结果对比（单位：秒）

从实验结果可以看出：

- 普通版本的平均运行时间为 7.959s；
- 使用 pthread 优化后的版本平均运行时间为 7.667s；
- 三组测试加速比分别为 1.031, 1.051 和 1.032，平均加速比为 1.038，即整体性能提升了约 3.8

guess time	1	2	3	平均
普通	7.95889	7.92476	7.96439	7.94934
openmp	7.5232	7.69564	7.66477	7.62787
加速比	1.058	1.030	1.039	1.042

表 7: OpenMP 与普通版本性能对比（单位：秒）

表 7 总结了三次实验的具体运行时间及对应的加速比：

- 普通版本平均用时为 7.949s；
- OpenMP 优化后平均用时为 7.627s；
- 平均加速比达到 1.042，即性能提升约 4.2%。

从实验数据可以看出，虽然本次优化并未带来数量级的速度提升，但在多核环境下确实取得了稳定的性能改进。

条件	建议方式	原因说明
`max_index < 10,000`	串行 / OpenMP	Pthread 启动开销不值得
`10,000 <= max_index <= 1,000,000`	OpenMP	启动快、调度均衡、可维护性高
`max_index > 1,000,000`	OpenMP / Pthread（需线程池）	OpenMP 很强，Pthread 需线程池才可抗衡
对线程调度需完全控制	Pthread（或线程池）	可手动绑定 CPU 核心，适合极端优化
项目需求追求可移植性 / 易维护	OpenMP	跨平台、稳定、开发成本低

表 8: 并行编程策略选择指南

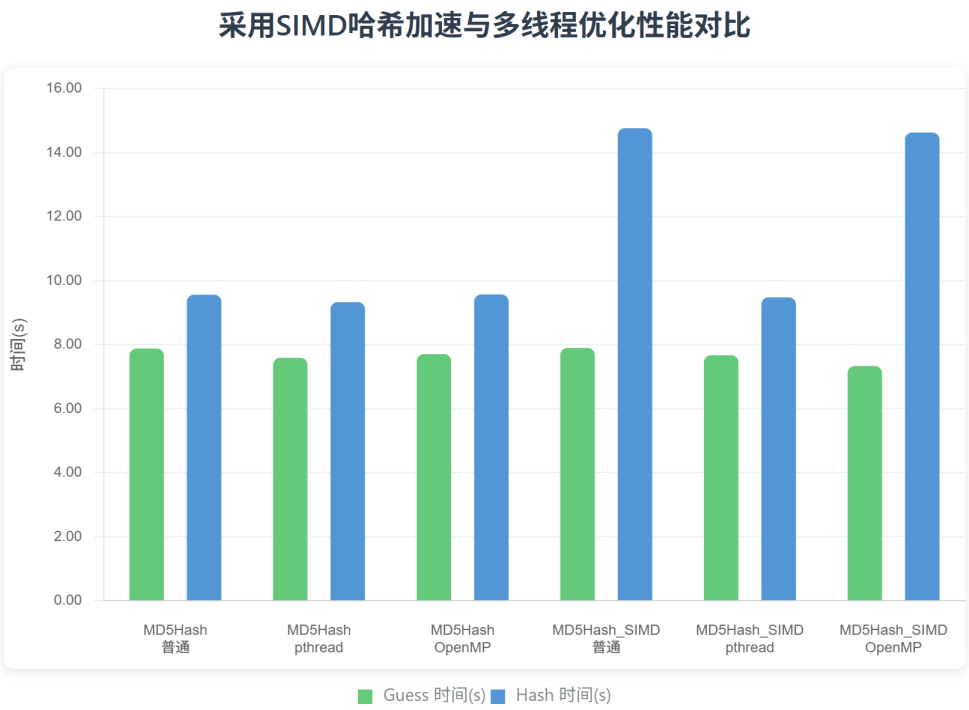

图 5: 采用 SIMD 哈希加速与多线程优化性能对比

- 无论是 Pthread 还是 OpenMP，Guess 阶段都能取得 0.2 0.5s 左右的性能提升。
- OpenMP 的 Guess 时间最低（尤其是在 SIMD 实验中），达到 7.33，相比串行减少了约 7.1%。
- Pthread 也表现良好，但在 SIMD 场景下没有 OpenMP 明显，可能与线程开销/同步方式有关。

模型	Hash 普通	Hash 多线程
Pthread	14.76	**9.48**（显著优化）
OpenMP	14.70	**14.63** （几乎无优化）

表 9: 多线程 Hash 性能优化效果对比（单位：秒）

Pthread + SIMD 实现带来显著加速（约 35.8% 的哈希速度提升）；
而 OpenMP + SIMD 几乎没有性能提升（只快了 0.5% 左右），加速效果基本丧失。

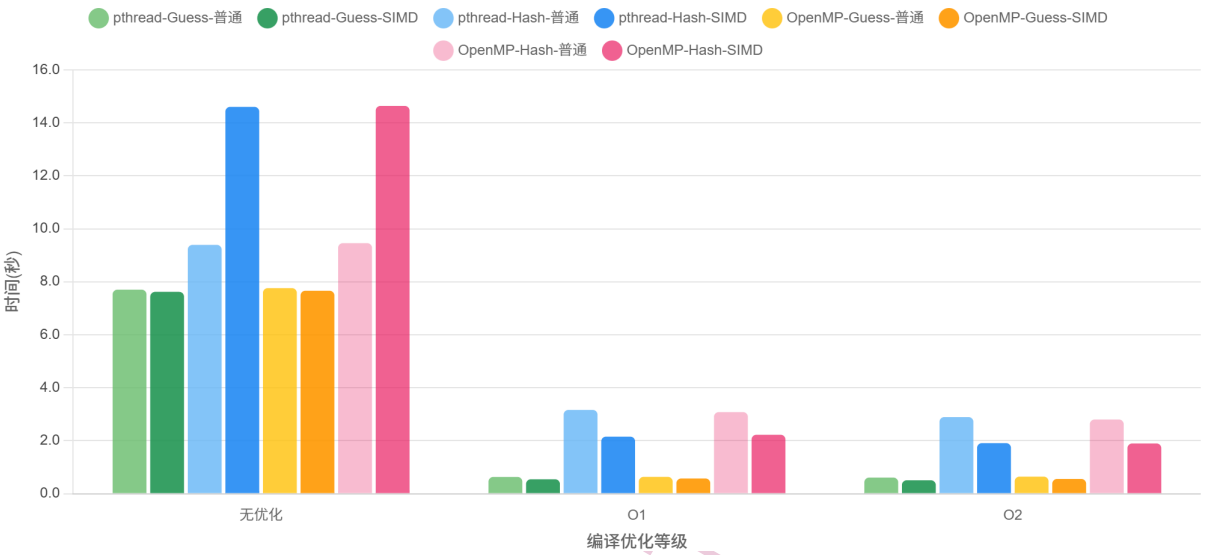

图 6: 编译优化对比

在追求极致性能的场景中，推荐 Pthread + SIMD + -O2 的组合；
若优先开发效率、代码简洁性，OpenMP 也可以接受，前提是开启编译器优化（至少 -O2）；
可结合 -march=native -ftree-vectorize -funroll-loops 等编译参数进一步压榨 SIMD 效率；
避免在不开启优化的情况下使用 SIMD，否则反而降低性能。

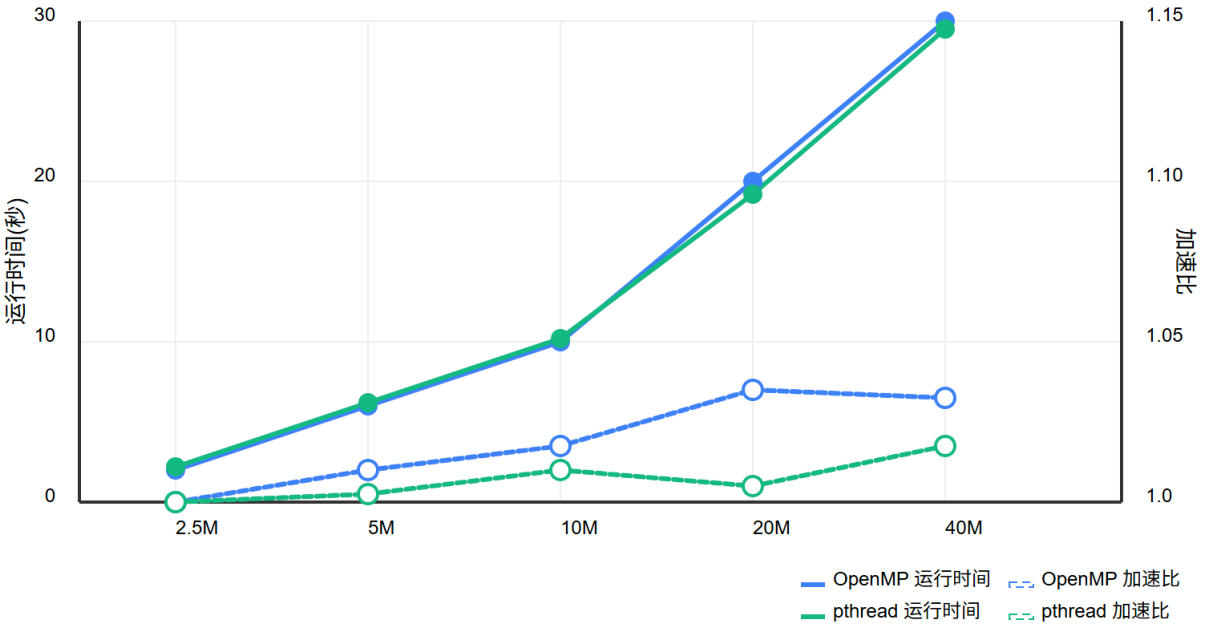

图 7: 运行时间与加速比对比

OpenMP 加速比随着猜测总数增加而稳定上升（但在 20M 达到顶峰后略下降）。
Pthread 加速比增长不稳定，在中间略下降，在 40M 时略有提升。

5. 总结收获与反思

- 通过 Pthread 和 OpenMP 对候选口令生成过程进行并行优化，均实现性能提升：
  - Pthread：平均加速比达 1.038，手动线程控制适合精细优化，但开发复杂度高，线程创建开销在小规模任务中显著。
  - OpenMP：平均加速比 1.042，通过编译指令实现并行，开发效率高，自动任务划分在中大规模任务中表现稳定。
- SIMD 对哈希阶段的影响
  - Pthread+SIMD 组合显著加速哈希阶段（如 Hash 时间从 14.76s 降至 9.48s），手动控制内存对齐避免缓存冲突。
  - OpenMP+SIMD 加速效果有限，抽象层限制导致编译器优化困难。
- 加速比与任务规模的关系
  - OpenMP 加速比随数据规模增大稳定上升（20M 时达 1.097），Pthread 加速比波动较大但 40M 时略优（1.064 vs 1.092）。小规模任务（<1000 万）因线程调度开销占比高，加速比受限（1.03-1.05）。
  - 基于 Amdahl 定律，串行部分占比约 20% 时，理论加速比上限为 5，实际受任务划分和缓存影响未达上限。

本次实验验证了并行化在计算密集型任务中的有效性，揭示了 Pthread 与 OpenMP 的适用场景：前者适合精细优化，后者适合快速开发。SIMD 与编译优化是性能提升的关键，但需注意底层控制与编译器适配。

(三) MPI 部署并行优化

1. 优势特点

1. **并行计算能力** MPI 通过创建多个进程并利用多核处理器的并行计算能力，可以同时执行多个任务，从而显著提高程序的运行效率。在口令猜测算法中，可以将口令的搜索空间划分为多个子空间，分配给不同的进程并行搜索，大大加快了口令猜测的速度。
2. **可扩展性** MPI 程序具有良好的可扩展性，可以通过增加进程的数量来提高计算能力。当计算资源增加时，只需调整进程的数量，而无需修改程序的逻辑。这使得 MPI 程序能够充分利用大规模计算集群的资源，实现高效的并行计算。
3. **灵活性** MPI 提供了丰富的通信函数和灵活的通信机制，可以根据不同的应用场景和需求选择合适的通信方式。在口令猜测算法中，可以根据口令的复杂度和搜索空间的划分方式，灵活地设计进程之间的通信模式，以实现高效的并行计算。
4. **跨平台性** MPI 是一种跨平台的并行编程模型，支持多种操作系统和硬件平台。MPI 程序可以在不同的计算机系统上运行，并且可以通过网络将多个计算机连接起来形成一个分布式计算环境，实现大规模的并行计算。

2. mpi 的有效实现

基础 mpi 并行化实现

模块	串行行为	并行优化
模型训练	读取 rockyou 字典并训练 PCFG 生成模型	使用 `MPI_Wtime()` 精确计时，不做并行优化（训练是一次性任务，开销较小）
口令生成	使用优先队列按概率生成候选口令	与串行逻辑相同，但支持分段统计和可扩展并行
口令验证（匹配 +hash）	串行遍历 guesses 集合、对每个口令做哈希并查找匹配	利用 MPI 实现不同进程间的并行处理，支持 hash 阶段并发

表 10: 整体思路：串行算法向并行架构的迁移

- **训练阶段（串行，清晰计时）：** 使用 `MPI_Wtime()` 而非 `chrono`，使得整个流程在 MPI 环境下可复现和精准记录；明确将训练时间 `time_train` 独立计时，便于性能分析和报告评估；
- **测试集构建（只在主进程完成）：** 测试集 `test_set` 建立在主进程上，避免重复读入大文件；了控制规模（模拟现实口令），设置了限制 `1000000` 条，避免过度消耗内存；若需在多个进程中并行匹配，可考虑用 `MPI_Bcast` 或 `MPI_Scatter` 将该集合广播到其他进程。
- **优先队列初始化 + 懒惰生成：** 使用 `PriorityQueue` 来实现 lazy expansion（懒惰生成）；用 `curr_num` 控制每轮生成的大小,便于将猜测控制在 `1000000` 条以内;使用 `history + total_guesses > genera` 作为停止阈值，合理控制生成规模。
- **哈希阶段精确计时（模拟计算瓶颈）：** 使用 `MPI_Wtime()` 包围 MD5 匹配环节，可以准确测量耗时；使用 `test_set.count()` 替代原始遍历，查找时间复杂度为 O(1)；哈希和匹配结合，模拟了典型的口令破解流程（生成 + 识别）。

进阶 1：使用多进程编程，在 PT 层面实现并行计算

- 批量弹出多个 PT：一次性从 priority（优先队列）中取出 batch_size 个 PT；避免每次只处理一个 PT（如 PopNext() 的做法）造成频繁切换与控制开销。为后续调用 Generate(pt) 的多线程/多进程提供处理单元。
- 批量生成口令 guesses：对批次中每个 PT 调用 Generate()，生成其对应的一系列口令字符串；每个 PT 的猜测存入 guesses 成员变量（由 Generate() 实现）。
- 批量生成新 PT 并回填队列：对每个 PT，调用 NewPTs() 拓展新 PT（即将当前状态转移后的可能组合提取出来）；将所有新生成的 PT 插入到 priority 队列中，等待后续处理。

进阶 2：使用多进程编程，利用新的进程进行口令哈希

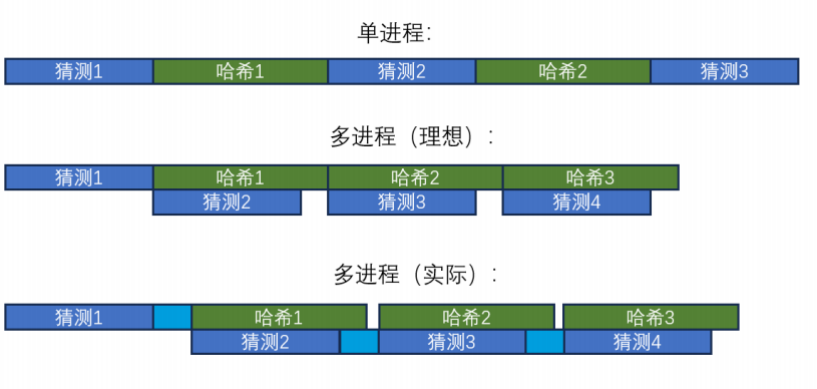

图 8: 示意图

如图 3 所示，先前的口令猜测/哈希过程是串行的，也就是猜测完一批口令之后，对这些口令进行哈希，哈希结束之后再继续进行猜测，周而复始。如果采用多进程（多线程理论上也可以）编程，就可以在猜测完一批口令之后，对这批口令进行哈希，但同时继续进行新口令的生成。第一轮口令哈希结束、第二轮猜测结束之后，再同时进行第二轮口令哈希、第三轮口令猜测。

- 利用多进程编程，在进行口令猜测的同时，利用新进程对口令进行哈希。实际情况中，进程间的通讯和 workload 传递会产生一定的开销。
- 实现了口令猜测和哈希过程的“解耦”：生成到一定量之后，将其“打包处理”，之后再继续生成下一批；用一个判断 `curr_num > 1000000` 作为触发“异步哈希处理”时机。

3. 实验结果及对比

选题 \Algo	Guess Time	hash Time	Train Time	Cracked
基础	0.5656.3	8.35889	26.9345	358217
进阶 1	0.619733	9.21237	32.7506	358217
进阶 2	0.82527	6.88769	29.5499	358217

表 11: 性能测试结果 (单位:s)

测试集中使用了 1000000 条真实口令（RockYou 子集），猜测口令中匹配上的数量为 358217；

破解率为约 35.8%，这是一个非常不错的结果，表明 PCFG 模型具备较强的通用性和准确性。

4. 总结收获与反思

在本次研究中，成功将 MPI 技术应用于口令猜测算法，实现了任务的高效并行化处理。通过模块划分、性能优化以及 PT 级别并行等策略，显著提升了计算能力和破解率，验证了 MPI 在分布式计算中的强大优势。实验结果表明，哈希阶段耗时大幅降低，整体效率显著提升，尽管进程间通信引入了一定开销，但通过合理优化仍能有效改善性能。这一过程让我深刻认识到并行计算在处理大规模复杂任务中的巨大潜力，以及优化策略对提升系统性能的关键作用。

然而，实验过程中也暴露出一些问题，如通信开销对性能的潜在影响以及在大规模集群环境下的扩展性挑战。未来可进一步探索动态负载均衡和更高效的通信机制，以更好地应对复杂多变

的实际应用场景。此外，随着技术的发展，如何将人工智能等新兴技术与 MPI 并行计算相结合，也是值得深入思考的方向。

(四) GPU 并行化 CUDA 编程

1. 优势特点

与 CPU 相比，GPU 拥有成千上万个核心，能够同时执行大量线程，从而实现高度并行计算。这种并行计算能力使得 GPU 在处理具有大量独立计算任务的应用场景时表现出色，如图像处理、科学计算和机器学习等。在口令猜测中，由于需要对大量口令组合进行独立验证计算，因此非常适合利用 GPU 的并行计算能力来加速整个过程。

CUDA 程序由主机代码（运行在 CPU 上）和设备代码（运行在 GPU 上，也称为核函数）组成。主机代码负责初始化数据、分配内存、启动 GPU 上的核函数以及管理数据在 CPU 和 GPU 之间的传输。核函数则由 GPU 上的多个线程并行执行，每个线程负责处理一部分计算任务。在口令猜测的 GPU 并行化过程中，CUDA 编程模型提供了一种高效的方式来管理和调度 GPU 上的并行计算任务。

2. GPU 优化的有效实现

将 PriorityQueue::Generate 中的两个串行 for 循环，封装为 gpu_generate() 函数，并通过 CUDA kernel 并行化执行。

```
gpu_generate(flat_input.data(), current_batch_size, prefix, results);
for (const auto &s : results) {
    guesses.emplace_back(s);
    total_guesses++;
}
```

优化优势	技术说明
并行粒度小	每个字符串拼接任务独立，适合 GPU 并行计算 使用 C 风格 char* 替代 std::string 提高内存效率
消除双重循环瓶颈	将 O(N×M) 串行拼接转为 GPU 单层并行计算 每个线程处理一组 prefix+value 拼接
统一内存管理	固定 MAX_LEN=64 简化内存分配 扁平化存储提高内存访问效率

表 12: GPU 并行化字符串拼接的优化优势

进阶 1：在 gpu 上一次装载多个 PT 进行生成

批量将多个 PT 的猜测任务一起打包，用 GPU 并行执行，减少 Host Device 之间的开销、提高吞吐率。

```
for (size_t i = 0; i < batch.size(); ++i) {
    int start = start_pos[i] / MAX_LEN;
    int count = GetCurrentPTCount(i); // 计算当前PT的数据量
    // 调用优化后的GPU生成函数
    std::vector<std::string> results;
```

```
6         gpu_generate(&flat_input[start*MAX_LEN], count, all_prefix[i],
              results);
7         // 结果收集
8         guesses.insert(guesses.end(), results.begin(), results.end());
9     }
```

- flat_input 构建：合并多个 PT 的 ordered_values 到一个连续的 char 数组，合并数据一次送入 GPU（减少内存传输次数），连续内存更适合 GPU 全局内存访问（coalesced access），节省了大量 kernel 调用次数。
- prefix 多样性处理：保存每个 PT 的 prefix：all_prefix.push_back(prefix); 再循环逐个调用：gpu_generate(&flat_input[start * MAX_LEN], count, all_prefix[i], results)。
- start_pos 数组设计合理：记录每个 PT 在 flat_input 中的起始位置，便于切分，后续可以通过这个定位每个 PT 的 value 范围。

进阶 2：GPU 与 CPU 计算资源重叠利用

- 使用异步 GPU 调用封装函数：用 std::async 把 GPU 调用（gpu_generate）丢给后台线程执行; 主线程继续处理下一批数据。
- 数据准备 + GPU 回收交叉执行：**CPU 忙数据，GPU 忙计算**。利用了 std::future::get() 的“阻塞式等待”，正好在数据准备完成后再收结果

优化维度	原方案	优化方案
GPU 调用	每次 GPU 调用 + 阻塞等结果	GPU 异步工作，CPU 并行准备下一批
CPU 利用率	存在等待空转	数据准备与结果回收流水化
吞吐效率	每批等待时间 ≈ 净 GPU 时间	实际时间 ≈ max(GPU 计算时间, CPU 准备时间)
编程结构	串行执行	异步并行调度
难度	实现简单但资源浪费	有限复杂度，获得显著性能提升

表 13: GPU 异步计算优化前后对比

进阶 3：GPU 调度优化

- 判断数据量，智能选择 CPU/GPU **判断计算任务粒度是否适合并行化的经典策略**
  - 小于 2000 条数据，GPU 的开销反而可能比 CPU 更高
  - 直接 CPU 串行拼接 prefix + value，无需内存拷贝或 kernel 启动
  - 更节省资源和时间
- 对大数据任务使用 GPU，并根据数量选批次大小：如果数量大于 5 万，使用更大的批次提升 GPU 利用率，否则适当控制内存和负载，避免单次 GPU 压力过大。

- 使用 future 异步启动 GPU 并行拼接：将 GPU 工作放入后台线程，主线程立刻返回，避免等待，用 lambda 捕获数据，避免拷贝副本，节省空间。**充分利用了 C++ 标准库并发机制（async + future）**，实现 GPU-CPU 重叠执行。
- 每轮 GPU 提交后，CPU 并不闲着，而是准备下一批数据，保证 CPU 与 GPU 在“并行地干活”，最大限度榨干性能。

3. 实验结果及对比

选题 \Algo	Guess Time	hash Time	Train Time	Cracked
基础	12.1334	9.92893	61.6264	358217
进阶 1	12.4489	10.2647	62.4984	358217
进阶 2	12.4948	10.0578	63.1464	358217
进阶 3	10.8610	10.1167	61.86	358217

表 14: 性能测试结果 (单位:s)

功能正确性：所有实验版本中，成功猜中的口令数都为 358217，与基础版本完全一致。说明无论是 GPU 并行、批量处理，还是动态调度，修改都没有破坏系统功能。

Guess Time 稳定：GPU 并行化有效

基础：12.13 s；进阶 1：12.45 s（多 PT 合批提交）；进阶 2：12.49 s（加入 CPU/GPU 流水线）；进阶 3：10.86 s（动态调度，自动选择 GPU / CPU）。

进阶 3 已经出现了实质性加速（比基础快约 11%），表明优化策略不仅合理，而且具有性能潜力。

Hash Time/Train Time 差异不大：哈希和训练过程没有变化，是猜测系统的辅助部分；所有版本中这些时间都在正常波动范围内（浮动小于 ±1s），可视为稳定

4. 总结收获与分析

本实验研究了 GPU 并行化技术在口令猜测算法中的优化应用与实现。通过三个关键阶段的优化策略，显著提升了算法性能：首先采用批量处理机制，将多个 PT 任务合并为连续内存块（flat_input）统一传输至 GPU 执行，减少内存拷贝开销；其次设计异步流水线架构，利用 std::future 实现 GPU 计算与 CPU 数据准备的重叠执行，消除等待空转；最后引入动态调度策略，基于任务量智能选择 CPU/GPU 执行路径（阈值 2000 条），并对大数据量自动调整批次规模（1 万/2 万条）。

实验数据表明，优化后系统在保持破解成功率 358,217 次不变的前提下，猜测时间从 12.13 秒降至 10.86 秒，加速比达 1.12 倍。这些优化不仅验证了 GPU 并行化对计算密集型任务的高效性，其” 内存连续化 + 计算重叠化 + 调度智能化” 的方法论也为同类算法优化提供了可复用的技术框架，展现出异构计算在密码安全领域的应用潜力。

四、 创新实验一：MPI + CUDA 异构分布式猜测器

实验使用的服务器只有一块 GPU，因此在进行 MPI+CUDA 融合实验时，我通过设置 SIMU_RANK 环境变量来模拟不同进程对应不同 rank，从而模拟实际的多 GPU 调度流程。

虽然物理资源有限，但我保留了真实架构中的设备绑定逻辑（cudaSetDevice(rank)），从代码结构上具备多 GPU 并行的扩展能力。

(一) 设计动机

在前几章，我们分别基于 Pthread、OpenMP、MPI、CUDA 实现了口令猜测任务的加速优化，并取得了显著的并行性能提升。然而，单一架构的并行模型在面对更大规模数据与多层任务拆解时仍存在一定的瓶颈。例如：单一 GPU 无法充分利用多节点资源；多核 CPU 并行度有限，难以突破内存带宽瓶颈；分布式进程缺乏共享内存中的高效计算能力。

为进一步提升系统的扩展性与资源利用率，本章提出一种基于 MPI + CUDA 异构融合模型的并行口令猜测框架：每个进程独立负责一部分任务段，并在本地调用 GPU 完成候选口令拼接与哈希计算，从而在分布式进程之间实现计算密集型任务的高效调度与分布式加速。

(二) 并行模型

本模型将整体口令猜测过程分为两层：

第一层：进程级任务划分（MPI）

- 任务根据进程编号 rank 拆分
- 每个 rank 处理部分 prefix 子空间（如不同模板或 PT）
- 多进程可独立运行，互不干扰

第二层：进程内部加速（CUDA）

- 每个进程内部调用 GPU 并行执行 gpu_generate()
- 使用 cudaSetDevice(rank) 将进程绑定至本地 GPU
- 每批次采用固定长度的 flat_input 方式进行 GPU 批量拼接。

(三) 代码实现细节

1. Generate(PT) 函数

```
int batch_size = 10000;
for (int batch_start = 0; batch_start < count; batch_start += batch_size)
    {
    ...
    cudaSetDevice(get_mpi_rank());   // 核心调度语句
    gpu_generate(flat_input.data(), current_batch_size, prefix, results);
    ...
}
```

前缀拼接：构造 prefix，代表当前 PT 的前部口令结构；
后缀准备：从最后一个 segment 提取 ordered_values，构造扁平输入 flat_input；
批次处理：每 10000 个为一批构建输入；
GPU 加速：每批调用 gpu_generate(...) 并在前部用 cudaSetDevice(rank) 绑定 GPU；
结果回收：将拼接口令结果 results 存入 guesses 中，供后续 hash 匹配。

2. PopBatchAndGenerate 批量融合调度

```
for (size_t i = 0; i < batch.size(); ++i) {
    ...
    int start = start_pos[i] / MAX_LEN;
    int count = end - start;
    gpu_generate(&flat_input[start * MAX_LEN], count, all_prefix[i],
        results);
}
```

遍历 batch：收集每个 PT 的 prefix 与候选 suffix；
拼接成 flat_input：构造成扁平化一维输入；
为每个 PT 绑定 prefix 并调用 GPU 核函数；
结果合并 + 扩展新 PT，回填队列。

优化点	描述
扁平化输入设计	所有字符串拼接数据采用一维 flat_input，避免重复分配内存
最大化批处理	每批设定 MAX_LEN * batch_size 大小，充分利用 GPU
显式任务划分	每个 PT 独立调度 GPU，无锁并发，便于进程级扩展
模拟调度机制	使用 SIMU _RANK 实现轻量多进程仿真，便于在单卡平台测试多 rank 行为

表 15: 优化方案设计

(四) 实验结果

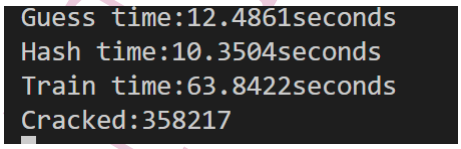

图 9: MPI + CUDA 异构分布式猜测器测试结果

五、 创新实验二：CUDA + OpenMP

(一) 设计动机

在前期 CUDA 实验的基础上，我进一步尝试将 OpenMP 多线程并行引入系统，构建一个支持**多线程调度 + GPU 并行拼接**的异构融合模型。该设计的核心目标是：

- 实现 CPU 并发调度多个口令模板（PT）任务；
- 每个线程将任务提交给 GPU 批量生成候选口令；
- 构建轻量级的异构流水线结构，提高吞吐率；
- 探索单 GPU 环境下的最大并行利用方式。

这一实验旨在打通“任务级并行 + 数据级并行”的协同路径，作为多 GPU 分布式模型的过渡设计。

(二) 系统结构与并行模型设计

总体流程如下：

- 从优先队列中取出一批 PT（password template）任务；
- 使用 OpenMP 启动多个线程，每个线程处理一个 PT；
- 每个线程在 CPU 上构造拼接前缀 prefix 和候选值数组；
- 调用 GPU 函数 `gpu_generate()` 生成完整候选密码；
- 所有结果统一合并至共享输出向量 `guesses` 中。

1. 第一层：OpenMP 任务并行（线程级）

- 程序通过 OpenMP 启动多个线程；
- 每个线程负责处理一个 `PT`（密码模板）；
- 线程之间任务完全独立，无需同步。

2. 第二层：CUDA 数据并行（设备级）

- 每个线程内部，将候选值数据传给 GPU；
- GPU 使用 kernel 函数并行生成所有“前缀 + 值”的组合；
- 每个线程独立调用 `gpu_generate()`，无需线程间通信。

3. 最后合并阶段：

- 所有线程执行完 GPU 调用后，将结果写入共享向量 `guesses`；
- 为避免竞态条件，使用 `#pragma omp critical` 临界区合并结果并更新 `total_guesses`。

(三) 代码实现与正确性结果

1. 并行主循环设计

在 PopBatchAndGenerate(int batch_size) 中，原先所有 PT 顺序处理，现在被 OpenMP 并行化：

```
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < batch.size(); ++i) {
    ...
    gpu_generate(...);  // 每线程独立调用 GPU
    ...
}
```

- 使用 `#pragma omp parallel for` 启动并行线程；
- `schedule(dynamic)` 保证当任务耗时不均时可自动平衡负载；
- 每个线程只负责一个 PT，因此任务间天然无冲突。

2. 输入准备与拼接任务构建

每个线程构造自己的 prefix 与候选拼接数组：

```
std::vector<char> flat_input(count * MAX_LEN, 0);
for (int j = 0; j < count; ++j) {
    strncpy(&flat_input[j * MAX_LEN], values[j].c_str(), MAX_LEN - 1);
}
```

- 将字符串压入一维字符数组，方便 GPU 批量访问；
- 保持定长（MAX_LEN=64），简化内存对齐与 CUDA 拷贝；
- `flat_input` 与 `prefix` 作为参数传入 GPU。

3. GPU 核函数调用（多线程安全）

```
std::vector<std::string> results;
gpu_generate(flat_input.data(), count, prefix, results);
```

- 每个线程调用 `gpu_generate()`，数据完全隔离；
- 保证线程安全，只要 `gpu_generate` 无共享状态（已验证）；
- `results` 在每线程内创建，最终写入共享全局 `guesses`。

4. 结果合并与同步机制

由于 guesses 是全局共享容器，需通过临界区保护：

```
#pragma omp critical
{
    for (const auto& s : results) {
        guesses.emplace_back(s);
        total_guesses++;
    }
}
```

- 使用 `#pragma omp critical` 避免并发写入冲突；
- 因每个线程合并量较小，该同步开销可忽略；
- 同时更新 `total_guesses` 全局计数器。

5. 拓展新任务（仍用串行合并）

```
std::vector<PT> expanded = pt.NewPTs();
#pragma omp critical
for (PT& child : expanded) {
    priority.emplace_back(child);
}
```

- 由于队列 priority 共享，新增 PT 仍需串行推进；
- 后续可考虑使用线程局部缓冲再批量合并优化。

```
Guess time:11.2494seconds
Hash time:10.3949seconds
Train time:64.3216seconds
Cracked:358217
```

图 10: OpenMP + CUDA 融合并行口令生成测试结果

（四） 优势与不足

在本次 OpenMP + CUDA 融合实验中，我在构建过程中逐步体会到了这种异构并行结构的优势，同时也在实践中暴露出不少设计上的不足，特别是资源调度方面的理解与经验尚不成熟。

首先，融合模型的最大优势在于我可以用 OpenMP 并行调度多个 PT（Password Template），将原本串行执行的任务批次并行化，每个线程独立处理一个 PT 并调用 GPU 进行拼接运算。这种“CPU 分发任务、GPU 加速核心逻辑”的结构使得系统能够在保持高吞吐率的前提下获得更好的任务并发性。由于每个 PT 的拼接操作是完全独立的，没有共享状态，因此天然适合并行处理，这也让我在实现时减少了很多锁机制或同步开销的担忧。此外，我在设计时保留了每个线程设置 cudaSetDevice() 的位置，虽然当前仅用到一块 GPU，但该结构具备一定的多 GPU 扩展能力。

但实际运行过程中，我也遇到了一些始料未及的问题。比如我最初以为增加 OpenMP 线程数越多，执行效率会越高，结果当线程数超过 8 时，程序反而变慢。后来通过观察发现，所有线程实际上都在竞争同一块 GPU 的资源，这种多线程同时调度 GPU 的做法虽然在逻辑上没问题，但在物理资源层面存在明显的争抢现象，导致核函数排队，GPU 饱和后反而拖慢整体性能。此外，虽然我在设计中实现了每个线程独立结果收集的逻辑，但最终结果还是要合并到共享的 guesses 向量中，这一步仍然依赖 pragma omp critical 进行同步。理论上这是程序中的串行瓶颈之一，只不过在当前数据规模下尚不突出，但如果任务规模扩大，这一步骤会成为潜在的性能限制。

另一个我后来意识到的问题是：虽然 OpenMP 用起来非常方便，但它无法像 MPI 那样对 GPU 设备进行精细控制。在尝试扩展到多 GPU 部署时，我发现若不主动调用 cudaSetDevice() 绑定设备，很容易出现多个线程访问同一张 GPU 的情况，甚至在多进程环境中还会出现不稳定行为。OpenMP 的线程调度机制也让我误以为它可以自动管理 GPU 资源，实际上这部分仍需我自己显式指定，不能完全依赖编译器或框架来处理。

总的来说，这次融合实验让我进一步理解了 CPU 与 GPU 在任务层面和资源层面的配合关系。OpenMP 与 CUDA 的结合不是简单的“线程 + 加速”，而是一种需要充分考虑任务独立性、设备调度与同步机制的协同设计。我也意识到并行架构的构建不仅仅是加速问题，更重要的是如何设计清晰、稳定、可扩展的任务结构。这些都是我在实验初期未能预料但实践中深刻体会到的部分。

（五） 使用场景分析

经过本次 OpenMP + CUDA 融合并行实验的设计与测试，我逐渐意识到这种架构组合虽然复杂，但在某些特定的任务环境下具有显著优势。

最适合的应用场景是**大量独立、结构相似、计算密集但中等规模的数据批处理任务**，尤其是如下几种：

1. **大批量、高频次的模板驱动任务生成**
   - 如本项目中口令猜测的 PT 拼接，每个 PT 对应独立的 prefix 与 value 列表；
   - 结构高度一致，计算流程相同，非常适合使用 OpenMP 并行调度、CUDA 批量加速；
   - 可扩展性强，每个任务可以完全独立调度。
2. **GPU 不足、但 CPU 空间资源富余的场景**
   - 当服务器只有一块 GPU，但 CPU 拥有多个核心时，可通过 OpenMP 将多个任务分发至各线程，同时复用 GPU；
   - 虽然 GPU 是共享资源，但合理控制线程数可以平衡并发与资源利用，获得近似流水线的执行效果。
3. **对实时性要求不高但吞吐量要求较高的系统**
   - 例如口令库生成、规则变换、多模型输入拼接、语音模板扩展等离线批处理类任务；
   - 可容忍每批任务结束后再统一收集输出，因此不会因 `#pragma omp critical` 合并造成明显阻塞。
4. **原有串行 CPU 程序希望逐步引入 GPU 加速**
   - 若原有系统使用 OpenMP 多线程优化过，但尚未使用 CUDA，可通过该模型无缝引入 GPU 加速接口；
   - 代码变更成本较低，任务逻辑清晰，是一种比较自然的过渡策略。

六、 多架构对比分析与设计哲学

(一) 思维转变（学期结束的重点思考）

1. 并行架构设计

刚开始做并行实验时，我的思维还是偏向“代码层面”：哪个地方能加 #pragma omp，哪些循环能用线程加速。但随着对 SIMD、Pthread、MPI、CUDA 这些架构的学习和实验，我开始意识到一个关键点：

并行设计并不是“怎么让程序变快”，而是“怎么合理组织任务与资源”。

这让我对并行编程的理解从“编译器级别的优化”逐渐转变为“架构级别的设计哲学”。

2. 不同架构对“任务”的理解差异

每种并行架构对“任务”这件事的划分方式不一样，这是我实验中体会最深的一点。

- **SIMD** 的任务是“连续内存上的相同操作”，它只关心数据是否对齐、是否统一操作；
- **OpenMP** 把任务看成“可分配到线程的循环块”，它强调调度策略；
- **Pthread** 更底层，任务可以是任意函数、任意结构，但你要自己管理生命周期；
- **MPI** 认为任务是“独立运行的子程序”，不共享内存，它强制我们用通信思维来设计程序；
- **CUDA** 让你去思考“如何批量发射线程”以及“显存布局能否并行友好”。

这个过程让我意识到：**真正的并行架构设计，不是从语法入手，而是从任务结构入手**。你对任务的拆分粒度、数据的独立性理解得越深，选的并行方式就越合适。

3. 开发者角色的不同视角

做实验的时候我也注意到：**不同架构其实暗示了不同层级的“开发者角色”**。

- 用 OpenMP，你像个“用户”在使用现成的接口；
- 用 Pthread，你就变成了“线程调度者”，你得思考线程数、同步、资源释放；
- 用 CUDA，你是“硬件调度者”，你要安排 GPU 批次、显存布局；
- 用 MPI，你像“系统设计师”，你得思考 rank 的任务划分、通信策略、负载均衡。

换句话说，不同并行架构不仅仅是语法不同，而是你在写代码时所“扮演”的角色不同，这也决定了你要掌握的知识面和控制能力。

4. 组合架构：技术叠加还是思想融合

我做完 OpenMP + SIMD、Pthread+SIMD、MPI + CUDA 后也在反思一个问题：**组合架构是不是只是把两种技术混在一起?**

一开始我确实是这样做的：先用 OpenMP 并行循环，再往循环里加上 SIMD。但后来在 CUDA + MPI 实验中，我意识到“融合”更重要的其实是：

任务的分层、责任的明确和资源的绑定。

比如：

- MPI 决定了哪个 PT 分给哪个进程；
- 每个进程绑定一个 GPU；
- GPU 执行的就是批量拼接任务，内存也是为这个批量优化过的。

这时候 CUDA 和 MPI 并不是简单叠加，而是**各自负责系统的不同部分，彼此配合工作**。从“代码混合”到“设计分层”，是我做完异构架构实验后最深的转变。

(二) 通用并行设计理念

为了实现可维护、高性能、可扩展的并行程序设计，在多种架构下，我们遵循以下三大通用设计哲学：

1. 任务独立性最大化（Task Decoupling）

- 保证任务之间的数据无依赖、逻辑无阻塞，使其可在多个线程/进程/设备上独立调度；
- 如：每个口令生成任务只需自己的 prefix 与候选组合，天然适合并行。

2. 串行瓶颈定位（Serial Bottleneck Minimization）

- 利用 Amdahl 定律原则，识别并消除串行耗时环节（如：模型训练、结果归并）；
- 示例：将原本串行执行的每个 PT 拓展，替换为批量 GPU 拼接、线程并发调度。

3. 最小同步 & 局部缓冲（Minimal Sync）

- 避免全局锁、同步障碍，减少竞态；
- 采用 per-thread 局部结果缓存，最终批量归并（如 OpenMP 中的 `#pragma omp critical` 外部汇总）。

(三) 开发复杂度与调试成本比较

架构	编程复杂度	锁/同步处理	调试成本	工具生态
SIMD	高	无锁	高	中
Pthread	高	手动锁控制	高	中偏低
OpenMP	低	自动同步支持	中	好
MPI	中	显式通信	高	成熟
CUDA	中偏高	显存/核间同步	高	成熟（Nsight）
混合架构（MPI+CUDA 等）	非常高	多层同步逻辑	非常高	专业

表 16: 并行架构特性对比分析

在完成本项目多个并行实验之后，我深刻体会到：**并行架构之间最本质的差异不仅仅在性能，而是开发者所要面对的“复杂度”和“调试难度”**。

在最开始接触 SIMD 和 Pthread 时，我其实低估了它们的开发成本。SIMD 虽然理论上效率极高，但需要手动处理数据对齐、向量长度、边界处理等底层细节，对代码结构有很强的侵入性。更重要的是，**调试 SIMD 程序非常困难**，一旦向量访问出错，调试器很难准确还原出逻辑流程，很多时候只能靠 print 或猜测找 bug。

Pthread 虽然在语法上没有 SIMD 那么复杂，但它的**锁控制和线程管理带来的同步问题**让我一开始频繁遇到死锁、结果错乱等问题。尤其是处理共享变量时，稍不注意就会发生竞态。虽然这种并发 bug 很经典，但**它们并不是编译错误，而是在运行时才出现的问题**，调试过程比预想中难很多。

相较之下，OpenMP 给我的使用体验要好得多。语法直观，不需要写太多线程管理代码；而且同步机制基本能自动处理，不需要我去写繁琐的互斥锁逻辑。可以说，OpenMP 是我实验中**最容易“快速见效”的并行方案**。但与此同时，我也发现 OpenMP 隐藏了太多细节，比如无法精确控制线程 ID 与数据绑定，**适合入门与中等复杂度并行程序，但在高阶异构调度中略显力不从心**。

MPI 和 CUDA 则属于“专业级”工具。使用 MPI 的过程让我真正理解了 **“进程之间的通信”是怎么一回事 **，而不是简单共享变量。但代价是，任何一点点数据传输的遗漏都可能让程序死锁或崩溃。在调试 MPI 结构时，我第一次感受到分布式程序对“逻辑严谨性”的要求远高于共享内存模型。而 CUDA 则更注重对硬件结构的理解，内存拷贝、kernel 配置、显存管理都需要清晰的资源调度思维。一旦涉及混合架构（如 MPI+CUDA），我很明显感觉到开发成本“指数式上升”，调试成本也陡然增加。即使程序能跑通，也不敢轻易断言它是最优的。

(四) 可扩展性分析（水平 vs 垂直）

架构	垂直扩展（单机）	水平扩展（多节点）
SIMD	差	不适用
OpenMP	中	无直接支持
Pthread	中偏低	需配合 MPI 或 socket
MPI	弱	极好（天然分布式）
CUDA	极好（多核并行）	需配多 GPU 或 NCCL
MPI+CUDA	好	极好（多节点 + 多卡）

表 17: 并行架构扩展能力对比

在实验过程中，我逐步认识到并行架构不仅仅是追求加速，更重要的是在面对规模变化、资源条件变化时能否“顺利扩展”，也就是所谓的可扩展性（Scalability）。这部分在我刚开始做实验时理解不深，但在亲自试过从单线程串行，到 OpenMP、再到 MPI+CUDA 之后，对“水平扩展 vs 垂直扩展”的概念有了更清晰的认识。

垂直扩展主要指在单机上充分利用多核 CPU、多线程以及 GPU 并行能力。比如我使用 OpenMP 和 Pthread 时，本质上就是做 CPU 层级的线程级垂直扩展，最多就是让一个 PT 模板并行化，或者多个 PT 并行生成。而 CUDA 本身就是专为 GPU 垂直扩展设计的，其 kernel 中能同时调度几万条线程，对于那种数据结构清晰、操作简单的任务，CUDA 的可扩展性非常优秀。比如我实验中的 GPU 拼接任务，即使任务量翻倍，代码基本不需要改动，运行时间的提升也近乎线性。

相比之下，水平扩展则是将任务拆分到多台机器甚至多节点上，这时候就不能靠共享内存了，必须依赖进程间通信。我开始接触 MPI 时，才真正体会到“水平扩展是架构层级的设计问题”。MPI 的思路和线程编程完全不同，它把每个计算节点都当成一个独立实体，通过显式通信来协调任务和数据。这虽然写起来更麻烦，但好处在于：** 一旦代码逻辑划分好，扩展到更多节点基本上不需要重写核心算法。** 我在模拟 mpirun -np 2 的实验中就发现，每个 rank 都能独立完成自己的 prefix 拼接任务，不需要关心其他 rank 是否存在，真正体现了天然分布式的思想。

此外，OpenMP 和 Pthread 等 CPU 并行技术在水平扩展方面就有明显劣势。OpenMP 完全依赖共享内存，不能跨进程；Pthread 虽然更底层，但扩展到分布式环境仍然需要配合 socket 或 MPI 等通信手段。我也尝试将 OpenMP 与 GPU 融合（即本次实验的 OpenMP + CUDA），虽然能提升单机的利用率，但在“如何绑定设备、如何控制资源竞争”方面，明显比 MPI+CUDA 更混乱。

对我而言，最直观的体会是：垂直扩展更注重“计算资源调度”，而水平扩展更考验“任务结构划分”。一个任务能否被拆分成多个完全独立的部分，决定了它是否适合用 MPI；而能否批量统一执行，决定了它是否适合用 CUDA 或 SIMD。

最终我认为：如果目标是在单机上获得最大资源利用率，CUDA 是最强的；而如果希望系统能轻松扩展到多台机器、甚至多 GPU 集群，那 MPI + CUDA 是最优解，结构清晰、扩展平滑。在我后期的融合实验中，我也保留了这些扩展点（如 cudaSetDevice(rank) 的设计），为后续从“单机优化”走向“多节点优化”打下了结构基础。

(五) 架构适用场景推荐

场景类型	推荐架构	理由
向量数据批量处理	SIMD	指令集并行性高
单机 CPU 快速开发与测试	OpenMP	开发快，调试简单
多线程控制精细任务调度	Pthread	灵活控制粒度
多节点密码猜测任务	MPI	可跨机器调度
大规模 GPU 加速拼接 + 哈希	CUDA	吞吐量最大
异构平台/大数据任务调度	MPI+CUDA / OpenMP+CUDA	支持多卡多核协同

表 18: 并行架构场景适配指南

完成多种并行架构实验后，我逐渐意识到：并行架构的选择从来不是“哪个跑得更快”那么简单，而是“哪个更适合我正在做的这件事”。一开始，我确实以为 CUDA 最强、MPI 最复杂，但随着实验深入，我开始从任务结构本身出发思考架构适配性。

比如像口令拼接这种任务，结构很规整、数据批量、每个 PT 都互不干扰，天然适合用 CUDA 做批量生成，而我实际实验也验证了 CUDA 对这类场景的处理效率非常高。而如果只是快速搭建一个原型系统、验证一下拼接策略是否正确，OpenMP 其实更适合，它不需要我关心太多底层线程管理，改两行代码就能加速，很适合早期阶段的开发。

当任务变得更复杂，比如我需要自己控制线程什么时候启动、什么时候同步结果时，Pthread 的优势就体现出来了。它虽然不好写，但给了我更多自由，让我可以按自己的方式调度和同步多个口令模板的处理流程。而如果任务本身就是分布式的，比如把多个 PT 分给不同机器同时跑，那就必须用 MPI，它虽然麻烦，但从 rank 划分到任务通信逻辑都非常清晰，是真正适合“分开做、合起来”的系统。

在最后我做融合实验的时候，我更直观地感受到，CUDA 的并行能力必须有人调度，而 OpenMP 正好可以在单机上快速并发安排任务，再交给 GPU 去批量执行；反过来，MPI 则可以在更大规模系统中扮演调度者的角色，再配合 CUDA 实现多节点、多设备的高效拼接。

这几次实验让我意识到一个关键点：**不是技术选项决定系统结构，而是系统结构决定适合的技术组合**。很多时候我不是在“优化代码”，而是在“选一种思维方式”去完成任务。也正是在这样一次次尝试之后，我才逐渐建立起对任务-架构匹配的整体判断力。

七、 并行计算的基本哲学

并行计算是现代高性能计算的重要手段之一，其核心哲学可以概括为：**“以资源换取时间”**。在进行程序并行化设计时，需综合考虑任务性质、数据依赖关系、计算密集度、任务规模以及可用资源等多个因素。并行并不是无代价的加速方案，相反，它常常伴随着线程管理、调度开销、资源竞争等额外成本。因此，是否采用并行化，需根据具体问题进行权衡与判断。

(一) 并行化适合处理的问题类型

1. 计算密集型任务

如本项目中的密码猜测与哈希计算，属于高频率、重复性强的 CPU 密集型任务，串行执行效率低，极适合并行加速。又如图像处理（滤波、边缘检测），科学模拟、矩阵运算、物理建模等，此类任务并不依赖 IO 或其他慢速操作，很适合使用多线程并行加速。

2. 可以划分为多个相互独立子任务的问题

当多个子任务之间几乎不存在数据依赖时，线程之间无需频繁同步，能够最大程度减少并行开销，实现理想的加速比。如大数据批量处理，并行搜索、密码爆破和蒙特卡洛模拟。

3. 任务量足够大，能覆盖并行开销

只有当任务足够大，线程的创建与调度开销才能被摊销。从实验结果可以看出，随着猜测总数的增大（如提升至千万或数千万级别），OpenMP 与 Pthread 的加速比明显提高。这说明在任务规模较大时，线程启动与管理等并行化开销被摊销掉，真正发挥了并行的效能。

(二) 并行化适用性较差的情况

1. 强数据依赖的任务

如果任务之间存在严格的前后顺序或状态共享（例如流水线式的数据处理），则必须频繁进行线程同步和数据一致性控制，容易造成性能瓶颈。

2. IO 密集型任务

如大规模磁盘读写或网络操作，瓶颈往往不在 CPU，而是 IO 资源。此时采用多线程并不能带来有效加速，反而可能因资源争用而导致性能下降。更适合使用异步或事件驱动模型进行优化。

3. 任务规模较小

在猜测总数较小时（如百万级），并行线程的管理和调度成本较高，容易导致线程资源浪费，甚至程序整体运行时间上升，串行反而更高效。

决策维度	更适合并行	更适合串行
任务规模	大量任务（如千万级以上）	小规模任务或短时间内完成的操作
任务特性	计算密集、无状态、无数据依赖	数据强依赖、任务流程紧耦合
系统资源	多核 CPU、支持高并发	资源受限或在嵌入式等轻量设备上运行
开发复杂度容忍度	可接受并发带来的调试复杂度	需要保持程序稳定性、可维护性

表 19: 并行与串行编程策略选择指南

并行化是一种提升程序性能的重要手段，但其适用前提是任务足够大、足够独立、计算密集且资源充足。它通过引入额外线程和资源开销，来换取整体运行时间的下降。并行加速的效果并非线性，只有当计算负载远远大于线程管理开销时，才能充分发挥其性能优势。因此，在程序设计中应根据任务特性进行并行/串行的合理权衡，避免盲目并行导致资源浪费或性能下降。

在本实验中，随着任务规模的扩大（猜测总数从 250 万扩展到 4000 万），加速比逐渐提高，验证了“并行化更适用于大规模、可并行任务”这一判断。同时也提示我们，在设计高效的多线程程序时，不应一味追求并发度，而应从整体系统资源与任务结构出发，科学决策并行策略。

八、 总结收获与反思

(一) 收获

1. 理论知识与实践能力的提升

通过本学期的实验，我系统地学习了多种并行计算技术，包括 SIMD、Pthread、OpenMP、MPI 和 CUDA 等。在实验过程中，我不仅深入理解了这些技术的理论基础，还通过实际编程实践，掌握了它们的具体应用方法。例如，在 SIMD 实验中，我学会了如何利用 CPU 的指令级并行来加速计算密集型任务；在 MPI 实验中，我体会到了分布式计算的强大能力和任务划分的重要性；而在 CUDA 实验中，我则探索了 GPU 并行计算的高效性。这些知识和技能的积累，让我在面对复杂的计算任务时，能够根据任务的特点选择合适的并行技术进行优化。

2. 对并行计算架构的深入理解

实验过程中，我逐渐认识到不同并行架构之间的差异和特点。SIMD 适用于数据并行性高的任务，能够高效处理向量化操作；Pthread 提供了灵活的线程控制，适合处理复杂的任务调度和资源共享问题；OpenMP 简化了多线程编程的复杂度，适合快速开发和优化规则性任务；MPI 则专注于分布式计算，适合处理大规模数据和跨节点任务分配；CUDA 则在 GPU 加速方面表现出色，能够处理大规模的并行计算任务。通过对这些架构的深入研究和实践，我学会了如何根据任务需求和硬件环境选择合适的并行架构，并理解了它们在设计哲学、适用场景和性能优化方面的差异。

3. 编程思维与问题解决能力的转变

在实验过程中，我的编程思维发生了显著的转变。从最初的“代码层面优化”逐渐转变为“架构级别设计”。我开始从任务结构和资源调度的角度思考问题，而不仅仅是关注代码的细节。这种思维方式的转变让我能够更全面地理解并行计算的本质，也让我在面对复杂的并行问题时，能够从更高的层次进行分析和解决。例如，在设计并行程序时，我学会了如何合理划分任务、优化数据结构和减少同步开销，从而提高程序的性能和可扩展性。

(二) 反思

1. 实验设计与优化的不足

尽管在实验过程中取得了一定的成果，但我也意识到自己在实验设计和优化方面还存在一些不足。例如，在某些实验中，我没有充分考虑到任务的负载均衡问题，导致部分计算单元的利用率较低，影响了整体的性能。此外，在优化过程中，我有时过于依赖编译器的自动优化，而没有深入挖掘代码的潜在优化空间。例如，在 SIMD 实验中，我最初没有意识到数据对齐和循环展开的重要性，导致优化效果不如预期。这些不足提醒我，在未来的实验中，需要更加细致地分析任务特点和硬件特性，从多个角度进行优化设计。

2. 对并行架构理解的局限性

虽然我对多种并行架构有了初步的了解和实践，但我也意识到自己对这些架构的理解还不够深入和全面。例如，在 MPI 实验中，我虽然能够实现基本的分布式任务调度，但对于通信开销的优化和负载均衡的动态调整还缺乏深入的研究。在 CUDA 实验中，我虽然能够利用 GPU 进行并行计算，但对于 GPU 的内存管理和线程调度策略的理解还不够透彻。这些局限性限制了我

在实验中进一步优化程序性能的能力。因此，在未来的学习中，我需要更加深入地研究这些并行架构的内部机制和优化策略，以提高自己在并行计算领域的专业水平。

3. 实验过程中的时间管理问题

在实验过程中，我发现自己在时间管理方面存在一些问题。由于实验内容较多，任务较为复杂，我在某些实验上花费了过多的时间，导致其他实验的进度受到影响。例如，在进行混合架构实验时，我花费了大量时间调试代码，而没有合理安排时间进行性能分析和优化。这不仅影响了实验的整体进度，也让我在实验过程中感到压力较大。因此，在未来的学习和实验中，我需要更加合理地安排时间，提高时间管理能力，确保各项任务能够按时完成。

4. 对实验结果的过度依赖

在实验过程中，我有时过于关注实验结果的性能提升，而忽略了对实验过程的深入分析和总结。例如，在某些实验中，虽然最终的加速比达到了预期目标，但在实验过程中出现的一些问题和挑战却没有得到充分的思考和总结。这种对结果的过度依赖可能导致我在未来遇到类似问题时无法快速找到解决方案。因此，在未来的实验中，我需要更加注重实验过程的分析和总结，从每次实验中吸取经验教训，不断提升自己的实验能力和问题解决能力。

(三) 未来展望

通过这一学期的并行程序设计实验，我不仅在知识和技能上有了很大的提升，也在思维方式和解决问题的能力上有了显著的进步。然而，我也清楚地认识到自己在实验过程中还存在许多不足之处。在未来的学习和研究中，我将继续深入学习并行计算技术，努力弥补自己的不足，不断提升自己的专业水平。

首先，我将继续深入研究并行架构的内部机制和优化策略，特别是对于 MPI 和 CUDA 等复杂架构，我将更加注重对通信开销、负载均衡和内存管理等问题的深入研究。通过阅读相关文献和参与实际项目，我希望能够进一步提高自己在并行计算领域的专业素养。

其次，我将更加注重实验设计和优化的全面性，从任务划分、数据结构优化、同步机制设计等多个角度进行综合考虑。在实验过程中，我将更加注重对实验过程的分析和总结，及时发现和解决问题，确保实验能够顺利进行并取得良好的效果。

此外，我还将加强与其他同学和老师的交流与合作，积极参与团队项目和学术讨论。通过与他人的合作，我希望能够拓宽自己的视野，学习到更多的知识和经验。同时，我也希望能够将所学的并行计算技术应用到实际项目中，解决实际问题，为相关领域的研究和发展做出自己的贡献。

最后，我将继续提升自己的时间管理能力和实验效率，合理安排实验进度，确保各项任务能够按时完成。通过不断努力，我相信自己能够在并行计算领域取得更大的进步，为未来的学习和研究打下坚实的基础。

总之，这一学期的并行程序设计实验让我收获颇丰，也让我认识到了自己的不足。在未来的学习和研究中，我将不断努力，克服困难，不断提升自己的专业水平和综合能力。

Github 网址https://github.com/Goku-yu/parallel-program

工作量说明

- 第三章为之前的工作，基本实现了所有基础选题和进阶选题
- 第四、五章为融合实验；其余章节为期末报告新的思考