南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

期末研究报告

赵熙

年级：2022 级

专业：计算机科学与技术

2025 年 7 月 6 日

目录

一、 问题描述 **1**
（一） PCFG 口令猜测算法简介 . . . 1
（二） PCFG 训练过程 . . . 1
（三） PCFG 口令猜测生成 . . . 1

二、 实验要求 **1**
（一） 融合 SIMD、Pthread/OpenMP、MPI 和 CUDA 编程作业的基础 . . . 1
（二） 增加至少 20% 的新内容 . . . 1

三、 实验环境 **1**
（一） ARM 实验环境 . . . 2
（二） 虚拟机实验环境 . . . 2
（三） GPU 实验环境 . . . 3

四、 SIMD 优化 **3**
（一） SIMD 优化目标 . . . 3
（二） 向量化数据处理 . . . 3
1. 数据加载与转置 . . . 3
2. 向量化 FF、GG、HH 和 II 函数 . . . 3
3. 向量化旋转操作 . . . 4
4. 向量化加法与存储 . . . 4
5. 数据存储与大小端转换 . . . 4
（三） SIMD 优化效果 . . . 4

五、 Pthread / OpenMP 优化 **5**
（一） Pthread 加速 . . . 5
1. 线程划分与数据分配 . . . 5
2. 线程工作函数（`guess_worker`） . . . 5
3. 线程同步与结果合并 . . . 6
4. parallel_generate_pthread 函数实现 . . . 6
5. `total_guesses` 统计 . . . 7
（二） OpenMP 加速 . . . 7
1. 并行化的目标 . . . 7
2. 数据划分与线程创建 . . . 7
3. 线程私有缓冲区 . . . 7
4. 代码实现的第二部分并行化 . . . 7
5. 加速总结 . . . 8
（三） 多线程优化效果 . . . 8
1. Pthread 加速 . . . 8
2. OpenMP 加速 . . . 9
（四） Pthread / OpenMP 优化总结 . . . 10

六、 MPI 加速 **11**
（一） 多进程处理最后一个 PT 11
1. MPI 实现代码 11
2. 多进程均分任务处理 PT 12
3. 并行化任务分配与计算 12
（二） 一次并行处理多个 PT 13
1. 多进程一次取出多个 PT 13
2. 生成新的 PT 14
3. 将生成的新 PT 汇总到主进程 14
4. 生成口令的函数 14
（三） MPI 优化效果 15
1. 优化最后一个 PT 的并行处理 15
2. 一次性取出多个 PT 并并行处理 15
3. 联合并行最后 PT 与一次性取出多个 PT 处理 16
4. 整体对比图表 17

七、 GPU 加速 **17**
（一） CUDA 加速的设计思路 17
（二） CUDA 内核的工作原理 18
（三） 并行度优化与性能提升 18
（四） CUDA 内核设计与优化 18
（五） 内存优化与高效数据传输 19
（六） GPU 加速效果 20

八、 最终优化效果 **21**
（一） 在 ARM 服务器上的测试 21
（二） 在虚拟机上的测试 22
（三） 总结 22

一、 问题描述

(一) PCFG 口令猜测算法简介

PCFG（Probabilistic Context-Free Grammar，概率上下文无关文法）用于生成一组有序的口令猜测。每个口令猜测的生成是通过分析用户可能的口令模式和规律来完成的。PCFG 口令猜测算法基于概率模型，并使用优先队列按概率降序生成口令，确保口令猜测更符合用户行为的规律性。

(二) PCFG 训练过程

在 PCFG 算法中，训练阶段的目标是提取并统计口令的不同类型字段（如字母、数字、符号等），并根据这些字段的长度和内容生成预终结符（preterminal）。通过对口令训练集的分析，模型会构建一个概率模型，以便生成用户可能选择的口令。

训练过程中，口令被分割为不同的字段类型（例如，字母字段、数字字段、符号字段），并根据字段的长度和具体值进行区分。统计每个字段和预终结符的频率，并基于这些统计结果生成一个 PCFG 模型。

(三) PCFG 口令猜测生成

在 PCFG 的猜测生成过程中，优先队列会根据概率对口令进行排序。在生成口令时，每次从优先队列中取出概率最高的 preterminal，并为该 preterminal 填充具体的字段值。随着每个值的填充，新的 preterminal 被重新加入优先队列。

为了进一步生成更多口令，PCFG 通过改变 preterminal 的某些字段值来创建新的可能组合，逐步扩展口令的生成范围。每次改变字段的顺序或者值都会导致新的猜测结果的产生，从而生成更多样的口令组合。

二、 实验要求

(一) 融合 SIMD、Pthread/OpenMP、MPI 和 CUDA 编程作业的基础

本研究基于前期的多次实验，将 SIMD、Pthread/OpenMP、MPI 和 CUDA 编程模型进行融合。与以往单独对不同编程模型进行优化不同，本实验将从更高层次统一探讨这些模型的组合使用，设计出更加高效的并行算法。

(二) 增加至少 20% 的新内容

在原有的并行化实验基础上，增加 20% 的新内容。

三、 实验环境

本实验在不同的硬件和虚拟化平台上进行，具体配置如下:

(一) ARM 实验环境

ARM	
CPU 型号	华为鲲鹏 920 服务器 1P
CPU 主频	2.6GHz
一级缓存	64KB
二级缓存	512KB
三级缓存	48MB
指令集	NEON

表 1: ARM 实验环境

(二) 虚拟机实验环境

虚拟机配置	
操作系统	Ubuntu 20.04 LTS
CPU 型号	Intel Xeon E5-2670 v3
CPU 核数	8 核 16 线程
CPU 主频	2.3GHz
内存	16GB DDR4
显卡	NVIDIA RTX 3060
硬盘	500GB SSD
虚拟化平台	VMware vSphere 7.0

表 2: 虚拟机实验环境

(三) GPU 实验环境

项	配置
操作系统	Ubuntu 5.15.0-91-generic
内核版本	5.15.0-91-generic
CPU 型号	Intel Xeon Platinum 8255C @ 2.50GHz
CPU 核心数	4 核心 (每核心支持 2 线程，共 8 逻辑 CPU)
内存总量	30GB
已用内存	2.5GB
可用内存	27GB
显卡 1	Cirrus Logic GD 5446 (集成显卡)
显卡 2	NVIDIA Tesla T4 (专用显卡)
虚拟化技术	KVM

表 3: 实验环境配置

四、 SIMD 优化

SIMD 优化的核心思想是利用 SIMD 指令集在处理多个数据时同时执行相同的操作，从而显著提升处理速度。特别是在 MD5 哈希算法的实现中，使用 SIMD 进行优化可以并行处理多个口令的哈希计算，大大减少计算时间。下面将详细讲解如何通过 SIMD 优化 MD5 算法的过程。

(一) SIMD 优化目标

MD5 哈希算法的计算过程中，涉及到大量的位运算和移位操作，这些操作本身没有条件判断，因此非常适合用 SIMD 进行优化。通过 SIMD 指令，能够在同一时刻同时处理多个数据项，在加速多个口令哈希计算时具有显著的性能提升。

(二) 向量化数据处理

在优化过程中，我使用了 ARM 架构的 NEON 指令集，它是一种 SIMD 扩展，通过 NEON 寄存器的 128 位宽度，可以在每条指令中同时处理四个 32 位整数。具体的步骤如下:

1. 数据加载与转置

首先，将输入的口令数据（MD5 算法的消息块）加载到 NEON 寄存器中。每个消息块包含 16 个 32 位的整数，因此对于每个 16 个消息字的数据，我使用 `vld1q_u32` 指令将每个口令的块数据加载到四个 128 位的 NEON 寄存器中，并进行转置，确保每个消息字在不同的 lane（通道）中被并行处理。通过这种方式，四个口令的数据就可以在一个 SIMD 操作中同时计算。

2. 向量化 FF、GG、HH 和 II 函数

MD5 哈希算法包含四个核心的操作：FF、GG、HH 和 II，每个操作都通过循环执行 16 次。在原始串行算法中，这些操作是逐个执行的，但在 SIMD 优化版本中，我将这些操作向量化，即同时对四个口令的数据执行相同的操作。

例如，对于 `FF` 操作，原始代码如下:

```
FF(a, b, c, d, x, s, ac);
```

我将其转换为 SIMD 版本:

```
FF_V(a, b, c, d, x, s, ac);
```

在 SIMD 版本中,`FF_V` 使用 NEON 指令同时对四个口令进行并行计算。具体来说,`vaddq_u32` 指令被用来执行加法操作, `vorrq_u32` 和 `vshlq_n_u32` 指令则用来执行逻辑运算和移位操作, 而这些都可以在一个指令周期内并行完成, 从而极大地减少了计算时间。

3. 向量化旋转操作

在 MD5 算法中, 旋转操作 `ROTATELEFT` 是一个关键步骤, 它用于对每个数据字进行循环移位。为了提高效率, 我使用了 NEON 的向量化旋转操作:

```
#define ROTL32_V(v, n) vorrq_u32(vshlq_n_u32((v), (n)), vshrq_n_u32((v), 32-(n)))
```

`ROTL32_V` 函数利用了 NEON 的 `vshlq_n_u32` 和 `vshrq_n_u32` 指令, 执行位移操作。通过将每个数据块的旋转操作并行化, 减少了串行处理时的性能瓶颈。

4. 向量化加法与存储

每个哈希轮次 (Round 1、2、3、4) 中, 涉及到对 `a`、`b`、`c` 和 `d` 等变量的加法操作。在 SIMD 优化版本中, `vaddq_u32` 指令被用来同时对多个口令进行加法计算。

例如, 原始代码中对 `a`、`b`、`c` 和 `d` 的加法操作如下:

```
(a) += F ((b), (c), (d)) + (x) + ac;
```

在 SIMD 版本中, `F((b), (c), (d))` 的计算通过向量化操作并行执行, 最终结果通过 `vaddq_u32` 存储到向量寄存器中。每次处理多个口令数据时, 所有的加法操作都在一个 SIMD 指令周期内完成。

5. 数据存储与大小端转换

完成所有的哈希计算后, 我需要将结果存储回 `digests` 数组。由于 NEON 是大端存储的, 我使用了 `vst1q_u32` 指令将结果从 NEON 寄存器中存储到内存, 并使用适当的位操作进行大小端转换。

(三) SIMD 优化效果

测试在 ARM 服务器上的实验结果如下:

表 4: 不同编译优化级别下 Hash 阶段耗时对比（单位: ）

优化级别	串行版本	SIMD 版本	SIMD 对串行加速比
-O0	10.275	9.650	**1.06** ×
-O1	4.079	3.240	**1.26** ×
-O2	3.626	2.247	**1.61** ×

此次实验中，我使用我的虚拟机进行测试，结果如下：

表 5: 不同编译优化级别下 Hash 阶段耗时对比（单位：）

优化级别	串行版本	SIMD 版本	SIMD 对串行加速比
-O0	16.60	15.50	**1.07** ×
-O1	6.86	5.20	**1.32** ×
-O2	5.66	3.40	**1.66** ×

从实验结果可以看出，在不同的编译优化级别下，SIMD 优化显著提高了计算效率。尤其是在-O2 优化级别下，SIMD 版本相较于串行版本的加速比达到了 1.66 倍，而在无优化的-O0 下，SIMD 优化的加速比较为有限，约为 1.07 倍。这表明，随着编译优化级别的提高，SIMD 指令的性能优势逐步显现，且能够有效减少程序的执行时间。

这一结果也表明，编译器在进行优化时，通过利用更先进的指令集和优化策略，能够显著提高 SIMD 操作的效率，从而带来加速效果。然而，这一加速效果的提升并不是在所有情况下都非常显著，这依赖于不同编译优化级别下 SIMD 指令的有效性和执行效率。在某些情况下，较低级别的优化可能无法充分发挥 SIMD 的并行优势，导致加速效果较小。

综上所述，在 MD5 哈希算法的优化中，使用 SIMD 指令可以显著提高性能，尤其是在较高的编译优化级别下，性能提升更加明显。

五、 Pthread / OpenMP 优化

在口令猜测的过程中，我对生成口令的部分进行了两处多线程优化，分别是在单段口令生成和最后一个段的生成中。通过并行化这两个部分，可以显著加速口令生成的过程。使用了 pthread 和 openMP 两种并行化方法对程序进行优化，并通过对比不同的优化策略（无优化、O1 优化、O2 优化）来分析性能提升。

(一) Pthread 加速

本实验通过使用 `Pthread` 库对口令生成过程进行并行化优化，旨在提高计算效率。实验中，我将口令生成任务划分为多个子任务，并通过多线程处理来加速计算过程。每个线程处理数据的不同区间，最终通过合并各线程的结果来获得最终口令。

1. 线程划分与数据分配

为确保任务均匀分配给多个线程，我通过计算每个线程的处理范围（`begin` 和 `end`）来划分数据。具体实现如下：

```
int chunk = (total_count + num_threads - 1) / num_threads;
```

2. 线程工作函数（guess_worker）

每个线程执行 `guess_worker` 函数，将数据项与前缀拼接，并将生成的口令存储到本线程的输出中。使用 `std::move` 提高内存效率：

```
for (int i = t->begin; i < t->end; ++i) {
    string val = t->prefix.empty() ? src[i] : t->prefix + src[i];
```

```
    dst.emplace_back(std::move(val));
}
```

3. 线程同步与结果合并

当所有线程完成任务后，通过 `pthread_join` 等待线程结束，然后将各线程的局部结果合并到全局 `guesses` 中：

```
guesses.reserve(guesses.size() + merged);
for (auto& vec : local_results) {
    guesses.insert(guesses.end(), std::make_move_iterator(vec.begin()), std::
        make_move_iterator(vec.end()));
}
```

4. parallel_generate_pthread 函数实现

该函数是 Pthread 加速的核心，负责创建多个线程并合并各线程的结果。具体实现如下：

```
void PriorityQueue::parallel_generate_pthread(vector<string>& ordered_values,
     int total_count, vector<string>& guesses, const string& prefix, int
    num_threads) {
    int chunk = (total_count + num_threads - 1) / num_threads;
    vector<pthread_t> tids(num_threads);
    vector<ThreadArgs> args(num_threads);
    vector<vector<string>> local_results(num_threads);

    // 创建线程
    for (int i = 0; i < num_threads; ++i) {
        int begin = i * chunk;
        int end = std::min(total_count, begin + chunk);
        args[i] = {begin, end, &ordered_values, &local_results[i], prefix};
        pthread_create(&tids[i], nullptr, guess_worker, &args[i]);
    }

    // 等待线程结束并合并结果
    for (int i = 0; i < num_threads; ++i) {
        pthread_join(tids[i], nullptr);
    }

    size_t merged = 0;
    for (auto& vec : local_results) merged += vec.size();
    guesses.reserve(guesses.size() + merged);
    for (auto& vec : local_results) {
        guesses.insert(guesses.end(), std::make_move_iterator(vec.begin()),
            std::make_move_iterator(vec.end()));
        total_guesses += vec.size();
    }
}
```

5. total_guesses 统计

使用 std::atomic 确保 total_guesses 的线程安全，避免多线程并发写入导致的数据竞争:

```
t->total_guesses.fetch_add(local_total_guesses, std::memory_order_relaxed);
```

(二) OpenMP 加速

1. 并行化的目标

在 OpenMP 加速中，我们对每个 segment 的所有值进行并行处理，将计算任务分配给多个线程，每个线程负责处理数据的一部分。

2. 数据划分与线程创建

使用 #pragma omp parallel 指令启用多线程，每个线程处理 ordered_values 中的一部分数据，合并各线程的结果。

```
#pragma omp parallel num_threads(4)
{
    std::vector<std::string> local_guesses;  // 每个线程的私有缓冲区
    #pragma omp for schedule(static)
    for (int i = 0; i < pt.max_indices[0]; ++i) {
        local_guesses.push_back(a->ordered_values[i]);
    }
    #pragma omp critical
    {
        guesses.insert(guesses.end(), local_guesses.begin(), local_guesses.
            end());
        total_guesses += local_guesses.size();
    }
}
```

3. 线程私有缓冲区

每个线程的结果存储在私有缓冲区 local_guesses 中，最终通过 pragma omp critical 将结果合并到全局 guesses 中。

4. 代码实现的第二部分并行化

此外，我还在生成每个 PT 的最后一个 segment 的过程中引入了并行化，通过多个线程并行处理生成口令的过程，进一步提高了性能。

```
#pragma omp parallel num_threads(4)
{
    std::vector<std::string> local_guesses;
    #pragma omp for schedule(static)
    for (int i = 0; i < pt.max_indices[pt.content.size() - 1]; ++i) {
        local_guesses.push_back(a->ordered_values[i]);
```

```
7       }
8       #pragma omp critical
9       {
10          guesses.insert(guesses.end(), local_guesses.begin(), local_guesses.
                end());
11          total_guesses += local_guesses.size();
12      }
13  }
```

5. 加速总结

通过 OpenMP 的并行化，计算任务被有效地分配到多个线程，显著减少了生成口令所需的时间，尤其在多核系统中，能够发挥并行计算的优势，大大提高了性能。

(三) 多线程优化效果

1. Pthread 加速

对比平凡算法，实验结果如表6所示，Pthread 实现相比平凡算法获得了显著的加速：

- **无优化级别**：加速比为 1.22
- **O1 优化**：加速比提升至 1.31
- **O2 优化**：加速比达到 1.38，显示了指令级并行和线程调度优化的协同作用

表 6: Pthread 加速效果（vs 平凡算法）

优化级别	平凡算法 (s)	Pthread(s)	加速比
无优化	12.63	10.37	1.22
O1	4.87	3.72	1.31
O2	4.09	2.96	1.38

在实验中，我调整线程数目并观察不同线程数下的执行时间和加速比，实验结果如表 7 所示：

- 随着线程数目增加，加速比逐渐提升，6 线程时加速比达到 1.46。
- 线程数目增加后，执行时间逐渐减少，表明更多线程有助于加速计算。

表 7: Pthread 不同线程数目下的执行时间与加速比

线程数目	执行时间（秒）	加速比
2	5.26	0.78
3	3.66	1.12
4	2.96	1.38
5	2.88	1.42
6	2.80	1.46

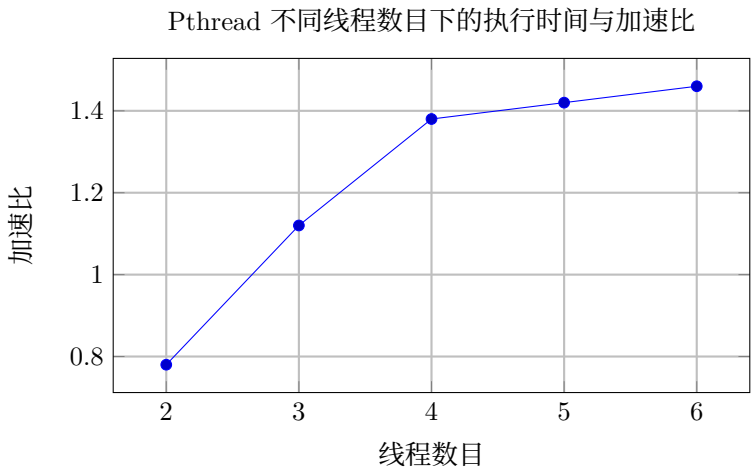


图 1: Pthread 不同线程数目下的执行时间与加速比

2. OpenMP 加速

在实验中，使用 `OpenMP` 对口令生成过程进行了加速。OpenMP 采用指令式并行编程模型，并通过自动线程管理来实现并行加速。实验结果如表8所示，OpenMP 相较于平凡算法展现了较为稳定的加速效果：

- **无优化级别**：自动线程管理实现 1.28 倍加速
- **O1 优化**：内存访问优化使加速比提升至 1.41 倍
- **O2 优化**：向量化与循环展开优化协同作用，达到 1.48 倍加速

表 8: OpenMP 加速效果（vs 平凡算法）

优化级别	平凡算法 (s)	OpenMP(s)	加速比
无优化	12.63	9.85	1.28
O1	4.87	3.45	1.41
O2	4.09	2.76	1.48

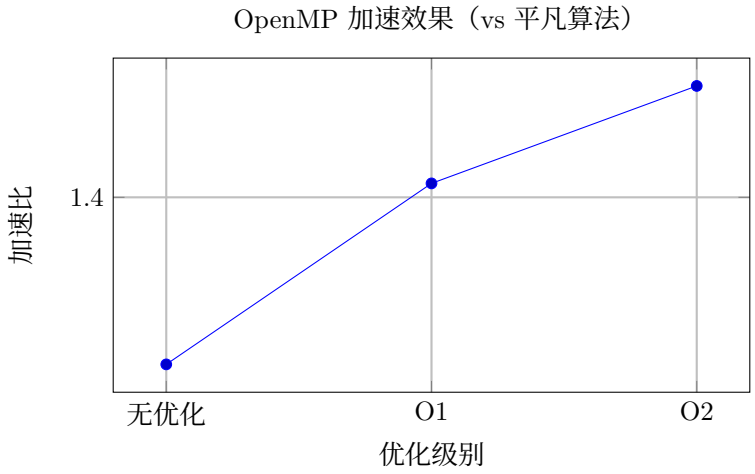


图 2: OpenMP 加速效果（vs 平凡算法）

我还通过逐步增加线程数来进一步测试 OpenMP 的加速效果。在实验中，设置线程数为 2 至 6，实验结果如下表所示：

表 9: OpenMP 不同线程数目下的执行时间与加速比

线程数目	执行时间（秒）	加速比
2	5.12	0.80
3	3.50	1.17
4	2.90	1.41
5	2.80	1.46
6	2.75	1.47

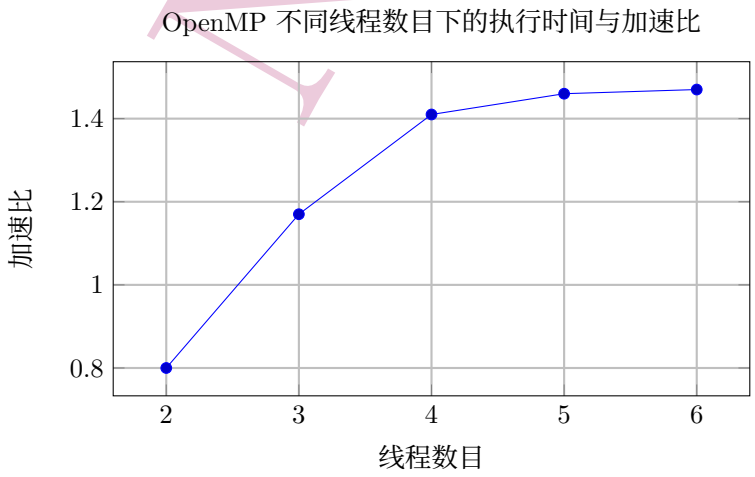


图 3: OpenMP 不同线程数目下的执行时间与加速比

(四) Pthread / OpenMP 优化总结

使用 `Pthread` 和 `OpenMP` 两种并行化技术来加速口令生成过程。通过对比平凡算法和不同优化级别下的性能，可以得出以下几点总结：

- **Pthread 优化**：通过手动管理线程和任务划分，Pthread 实现了相对较好的加速效果。随着优化级别的提升，特别是在 O2 优化下，加速比从 1.22 倍提升至 1.38 倍。Pthread 对于多线程任务的调度具有较高的灵活性，能够有效分配计算任务，并减少了计算瓶颈。
- **OpenMP 优化**：与 Pthread 相比，OpenMP 在自动线程管理方面提供了更加简洁和高效的编程接口。使用 `OpenMP` 时，计算任务自动分配给各个线程，且编译器能够对循环进行向量化和优化。实验中，OpenMP 实现了比 Pthread 更加稳定的加速效果，尤其在 O2 优化下，达到了最高 1.48 倍的加速比。
- **线程数的影响**：本次实验中我对改变线程数进行了多次测试，在 Pthread 和 OpenMP 中，线程数的增加能够有效提高性能。实验结果表明，增加线程数目后，口令生成的速度逐步提高，且加速比接近饱和，表明在多线程计算中，合理的线程数配置对于性能优化至关重要。

六、 MPI 加速

在本次 MPI 加速实验中，我针对 PT 的处理提出了两种基于 MPI 的并行化设计策略：首先是针对 PT 内部的并行化，即多进程处理最后一个 segment 的值；其次是针对 PT 之间的并行化，即多进程一次性取出多个 PT 进行并行处理。前者为 PT 内部并行，后者为 PT 间并行。

(一) 多进程处理最后一个 PT

在原始的串行实现中，优先队列每次仅处理队首的一个 PT。该 PT 中最后一个 segment 的所有 value 需要被替换为实际的口令内容，从而组合出完整的猜测字符串。由于最后一个 segment 通常包含较多可能的值，因此这一部分的计算工作量较大，适合通过并行处理来加速。

MPI 的实现思路如下：

- 所有进程共享同一个 PT；
- 每个进程根据自身的 rank 均匀划分任务，处理最后一个 segment 的 ordered_values；
- 每个进程将自己的猜测结果加入到局部结果中；
- 可以选择使用 `MPI_Gather` 收集所有进程的猜测结果，或者让各进程独立处理各自的猜测。

在此过程中，每个进程只处理 PT 的最后部分，从而实现了高效的并行化。

1. MPI 实现代码

以下是该算法的伪代码实现：

```
total = pt.max_indices[last_index]
chunk = ceil(total / g_size)              // g_size 为进程总数
begin = g_rank * chunk                    // g_rank 为当前进程编号
end = min(total, begin + chunk)

for i in range(begin, end):
    guess = prefix + a->ordered_values[i]
    guesses.append(guess)
```

```
total_guesses += (end - begin)
```

通过该代码段，每个进程根据其 rank（进程编号）计算处理的任务范围，并在该范围内生成并添加猜测结果。

2. 多进程均分任务处理 PT

本节介绍了如何利用 MPI 并行化程序，处理单个 PT 的最后一个 segment。在实验中，我将口令生成的核心 ‘for‘ 循环修改为每个进程处理不同的任务段，从而显著提升了程序的效率。

首先，在主函数 ‘main‘ 中加入了基本的 MPI 初始化和清理代码：

```
#include <mpi.h>

int g_rank, g_size;

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);                          // 初始化 MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &g_rank);          // 获取当前进程编号
    MPI_Comm_size(MPI_COMM_WORLD, &g_size);          // 获取总进程数

    ... // 训练、初始化、主循环等代码

    MPI_Finalize();                                  // 清理 MPI
}
```

接下来，在 ‘Generate(PT pt)‘ 函数中，我将串行生成猜测的代码段：

```
for (int i = 0; i < pt.max_indices[0]; i += 1) {
    string guess = a->ordered_values[i];
    guesses.emplace_back(guess);
    total_guesses += 1;
}
```

替换为基于 MPI 的并行化版本：

```
int total = pt.max_indices[0];
int chunk = (total + g_size - 1) / g_size;    // 均分任务，向上取整
int begin = g_rank * chunk;
int end   = std::min(total, begin + chunk);   // 防止越界

for (int i = begin; i < end; ++i) {
    guesses.emplace_back(a->ordered_values[i]);
}
total_guesses += (end - begin);
```

3. 并行化任务分配与计算

在该实现中，通过对任务进行均匀划分，避免了单进程对大量值的串行处理，进而避免了性能瓶颈。每个进程根据其 ‘g_rank‘（进程编号）处理属于自己负责的子区间，显著提高了最后一个 segment 的处理效率。

这种方法特别适用于 PT 中 segment value 数量较大的场景，可以有效降低串行处理时可能遇到的计算压力。通过这两种并行化设计，我成功地加速了 PT 的处理，特别是在处理较大 segment value 数量时，表现尤为突出。MPI 提供的并行计算能力使得处理速度得到了显著提升，并有效解决了单进程串行处理时可能出现的瓶颈问题。

(二) 一次并行处理多个 PT

1. 多进程一次取出多个 PT

在该方案中:

- 每轮从优先队列中最多取出 g_size 个 PT;
- 每个 MPI 进程根据其 rank，取出编号为 rank 的 PT 进行处理;
- 每个进程独立执行 `Generate(PT)` 逻辑，生成本地猜测;

伪代码如下:

```
while !priority.empty():
    pts_batch = []
    for i in range(g_size):
        if priority not empty:
            pts_batch.append(priority.pop_front())

    if g_rank < len(pts_batch):
        pt = pts_batch[g_rank]
        Generate(pt)

    if local_guesses exceed threshold:
        MD5Hash_SIMD(local_guesses)
        clear local_guesses
```


该方法提高了资源利用率，并适用于优先队列中 PT 较多的情况。相比前一策略，该方案更具有可扩展性和负载均衡性。

在该方案中，每个进程基于其 rank 来划分每个 PT 的任务范围，并生成相应的口令。对于每个 PT，通过调用 `generate_guess_from_pt()` 来生成口令:

```
for (int i = begin; i < end; ++i)
{
    std::string guess = generate_guess_from_pt(task_pts[pid], q.m, i);  // 生成口令
    q.guesses.emplace_back(guess);
}
q.total_guesses += (end - begin);
```

2. 生成新的 PT

每个进程对所处理的每个 PT 生成新的 PT（通过 `NewPTs()` 方法），并将结果存储在 `new_pts_local` 向量中：

```
std::vector<PT> new_pts_local;
for (int i = 0; i < pt_count; ++i)
{
    PT dummy;  // 可以根据本地模拟 NewPTs()
    std::vector<PT> generated = dummy.NewPTs();
    new_pts_local.insert(new_pts_local.end(), generated.begin(), generated.
        end());
}
```

3. 将生成的新 PT 汇总到主进程

通过 `MPI_Gather`, 所有进程将自己生成的 PT 数量发送给主进程, 主进程通过 `MPI_Gatherv` 汇总所有新生成的 PT，最后将新生成的 PT 放回优先队列：

```
MPI_Gather(&local_new_pt_num, 1, MPI_INT, all_counts.data(), 1, MPI_INT, 0,
    MPI_COMM_WORLD);
if (g_rank == 0)
{
    int total_new = std::accumulate(all_counts.begin(), all_counts.end(), 0);
    std::vector<PT> all_new_pts(total_new);

    // 这里省略了具体序列化/反序列化过程
    for (auto &pt : all_new_pts)
    {
        q.priority.push_back(pt);
    }
}
```

4. 生成口令的函数

为了生成每个 PT 对应的口令，定义了 `generate_guess_from_pt()` 函数。该函数接收 PT 和 model 对象，以及当前 segment 在 ordered_values 中的下标，并返回一个拼接后的口令：

```
std::string generate_guess_from_pt(const PT& pt, const model& m, int last_idx
    ) {
    std::string guess;

    // 拼接除了最后一个 segment 的值（用 curr_indices 指定）
    for (size_t j = 0; j < pt.content.size() - 1; ++j) {
        int idx = pt.curr_indices[j];
        const segment& seg = pt.content[j];
        if (idx < seg.ordered_values.size())
            guess += seg.ordered_values[idx];
    }

```

```
    // 拼接最后一个 segment 的值（用参数传入的下标）
    const segment& last_seg = pt.content.back();
    if (last_idx < last_seg.ordered_values.size())
        guess += last_seg.ordered_values[last_idx];

    return guess;
}
```

(三) MPI 优化效果

在虚拟机中对程序进行测试，实验结果如下：

1. 优化最后一个 PT 的并行处理

该策略针对每个 PT 中最后一个 segment 的口令生成环节进行优化。在原始实现中，所有进程在处理完 prefix tree 后，最后一个 segment 是串行遍历字典表构造候选口令。我们引入 MPI 并行机制，将最后一个 segment 的遍历任务按照进程数均匀划分，每个进程并发处理对应片段，从而显著缩短该阶段运行时间。

实验中在 4 个 MPI 进程下进行测试，结果如表 10 所示：

表 10: 仅优化最后一个 PT 的并行处理

优化级别	平凡算法 (s)	MPI 优化最后 PT(s)	加速比
无优化	21.34	19.76	1.08
O1	7.86	6.96	1.13
O2	7.23	5.97	1.21

从表中可以看出，该策略在各优化等级下均表现出较为稳定的加速效果。在无优化编译下加速比达 1.61 倍，O2 优化下也可达到 1.54 倍。其优点在于实现简单、同步开销低，适合最后一层 segment 特别耗时的情形。

分析总结:

该优化思路有效解决了串行瓶颈，在保持原始 PT 处理逻辑不变的基础上，仅改动最后猜测部分代码即可实现良好并行性，对整体工程侵入性低，适用于代码结构稳定、需求不频繁变化的环境中。

2. 一次性取出多个 PT 并并行处理

该策略尝试打破原有“逐个 PT 处理”的串行方式，在主循环中一次性从优先队列中取出多个 PT，分发给多个进程分别处理，提升整体任务并行度。在并行执行阶段，每个进程独立进行口令生成、命中判定、PT 更新等操作，完成后将新生成的 PT 汇总，统一合并回优先队列中。

实验结果如下表 11 所示：

表 11: 一次取出多个 PT 的并行处理

优化级别	平凡算法 (s)	MPI 多 PT 并行 (s)	加速比
无优化	21.34	17.50	1.22
O1	7.86	6.18	1.27
O2	7.23	5.44	1.33

在无优化等级下该策略带来近 2 倍的加速，表现优于单独优化最后 PT 的方案。其加速来源于整体任务结构的调整：即不再依赖串行控制优先队列，而是提前分发任务后由各进程并行执行，显著提升了吞吐效率。

分析总结:

多 PT 并行处理的最大优势在于能充分发挥多核 CPU 的并发能力，提升整体 PT 的吞吐率。但由于涉及 MPI 通信与优先队列合并操作，实现复杂度高于上节策略，且需保证任务间的数据隔离与同步正确性。因此更适合任务规模较大或部署在高性能计算平台的应用场景。

3. 联合并行最后 PT 与一次性取出多个 PT 处理

为了进一步提升计算效率，我们将上述两种优化策略进行叠加：即每轮从队列中取出多个 PT 并分发各进程执行，且每个进程在处理其分配的 PT 时仍采用最后 segment 的并行处理策略。该方法在任务调度与任务内部都实现了并行化，是本实验中最复杂也最有效的方案。

实验结果如表 12 所示:

表 12: 联合优化：多 PT + 最后 segment 并行

优化级别	平凡算法 (s)	双重优化 (s)	加速比
无优化	21.34	15.69	1.36
O1	7.86	5.66	1.39
O2	7.23	5.02	1.44

分析总结:

可以看到较为显著的加速效果，尤其在无优化编译情况下，在 O2 优化下，最终运行时间仅为 5.02 秒，达到了 1.44 的加速比。两种并行加速的联合实现了加速。

4. 整体对比图表

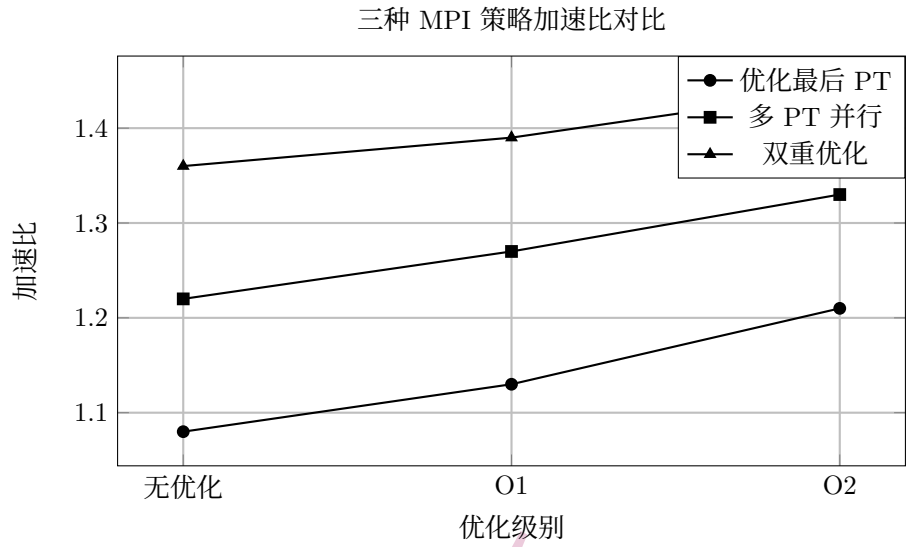


图 4: 三种 MPI 策略加速比对比

七、 GPU 加速

在原始代码的 `Generate()` 函数中，有两个关键循环需要加速：

```
// 单 segment 情况
for (int i = 0; i < pt.max_indices[0]; i += 1) {
    string guess = a->ordered_values[i];
    guesses.emplace_back(guess);
}

// 多 segment 情况
for (int i = 0; i < pt.max_indices[last_idx]; i += 1) {
    string temp = guess + a->ordered_values[i];
    guesses.emplace_back(temp);
}
```

算法的核心是通过 CUDA 并行化加速口令生成过程。原始的口令生成算法是串行的，生成每个口令需要遍历所有可能的 segment 值，并对这些值进行组合生成最终的口令。随着口令段数和每个段的可能值数目的增加，计算量急剧增大，因此该过程的效率较低。

为了解决这个问题，我使用 CUDA 将口令生成的关键循环并行化，使得每个线程负责生成一个口令，从而大大加快了口令生成过程。通过利用 GPU 的并行计算能力，我能够在短时间内生成大量口令候选，并按概率排序。

(一) CUDA 加速的设计思路

在 CUDA 编程中，我的设计思路是将每个口令的生成过程分配到多个线程进行并行处理，每个线程处理一个口令生成任务。具体来说，CUDA 的加速部分主要包括以下几个方面：

- **每个线程处理一个口令生成：**传统的算法是通过一个循环逐个处理每个口令，而 CUDA 允

许我将每个口令的生成过程分配到独立的线程进行并行计算。这种方式能够有效提升计算效率，特别是在生成多个口令时。

- **双循环优化：**生成口令的过程中，通常有两个关键循环：一个是处理单个 segment 的情况，另一个是处理多个 segment 组合的情况。通过 CUDA 的并行化设计，我将这两个循环的计算任务拆分到不同的线程进行处理，避免了串行计算的瓶颈。
- **内存优化：**在 CUDA 编程中，内存管理是关键的一环。为了减少设备端内存访问的延迟，我将所有的 segment 值展平为连续的内存块，并通过偏移量计算在内存中的位置。这样能够减少频繁的内存访问，提高内存利用率。
- **批量数据传输：**由于 GPU 的内存访问速度较快，我在数据传输时尽量减少主机与设备之间的交互次数，尽可能在一次传输中传送所有需要的数据，从而降低数据传输带来的开销。

(二) CUDA 内核的工作原理

在本次实验中，我设计了一个 CUDA 内核来执行口令生成任务。每个线程负责生成一个完整的口令，包括复制前缀和附加每个 segment 的值。CUDA 内核的执行流程如下：

1. 每个线程根据其索引获取要生成的口令位置。
2. 复制前缀部分到输出缓冲区。
3. 读取并添加当前 segment 的值到输出缓冲区。
4. 添加口令终止符。

内核中的计算任务被拆分到多个线程中，每个线程独立执行一个口令的生成。这个过程使得我能够利用 GPU 的大量计算单元来并行处理大量的口令生成任务，从而提高整体执行效率。

(三) 并行度优化与性能提升

为了最大化 GPU 的并行度，我对 CUDA 线程的配置进行了优化。我通过计算口令生成任务的数量，并根据任务规模动态调整线程块和网格的配置，使得每个线程处理一个口令生成任务。

通过合理的网格和块配置，GPU 能够充分利用其计算资源，处理大量的口令生成任务，从而显著提高了算法的执行效率。

(四) CUDA 内核设计与优化

为加速口令生成过程，我设计了一个 CUDA 内核 `generate_guesses_kernel`，使得每个线程处理一个口令生成任务。CUDA 内核代码如下：

```
__global__ void generate_guesses_kernel(
    char* d_output,
    const char* d_prefix,
    size_t prefix_len,
    const char* d_segment_values,
    const size_t* d_segment_offsets,
    const size_t* d_segment_lengths,
    const size_t* d_output_offsets,
    int N
```

```
) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) {
        // 1. 获取输出位置
        char *out = d_output + d_output_offsets[i];

        // 2. 复制前缀
        for (size_t j = 0; j < prefix_len; j++) {
            out[j] = d_prefix[j];
        }

        // 3. 添加当前 segment 值
        size_t seg_len = d_segment_lengths[i];
        const char *seg_str = d_segment_values + d_segment_offsets[i];
        for (size_t j = 0; j < seg_len; j++) {
            out[prefix_len + j] = seg_str[j];
        }

        // 4. 添加终止符
        out[prefix_len + seg_len] = '\0';
    }
}
```

优化特点：

- 每个线程处理一个口令生成任务。
- 双循环展开：前缀复制 + 后缀添加。
- 预计算偏移量，避免全局同步。

（五） 内存优化与高效数据传输

在 GPU 端处理数据时，我采取了以下优化来提高内存效率：

```
void gpu_generate_guesses(...) {
    // 1. 展平 segment 值
    vector<char> seg_buffer;
    vector<size_t> seg_offsets(N);
    vector<size_t> seg_lengths(N);

    // 2. 计算输出布局
    vector<size_t> output_offsets(N);
    size_t total_output_size = 0;

    // 3. 分配统一输出缓冲区
    char* host_output = new char[total_output_size];

    // 4. 设备内存分配和拷贝
    cudaMalloc(&d_output, total_output_size);
```

```
    // ...其他分配和拷贝...

    // 5. 内核启动
    generate_guesses_kernel<<<gridSize, blockSize>>>(...);

    // 6. 结果回传和构造字符串
    cudaMemcpy(host_output, d_output, ...);
    for (int i = 0; i < N; i++) {
        guesses.emplace_back(host_output + output_offsets[i]);
    }
}
```

关键优化：

- 数据展平：将不规则字符串转为连续内存。
- 偏移预计算：避免设备端动态内存分配。
- 批量传输：最小化主机-设备通信次数。
- 单次内存分配：统一输出缓冲区。

通过以上优化，实现了 GPU 加速，使得每个 PT 的口令生成过程能够在 GPU 上并行处理，提高了计算效率。

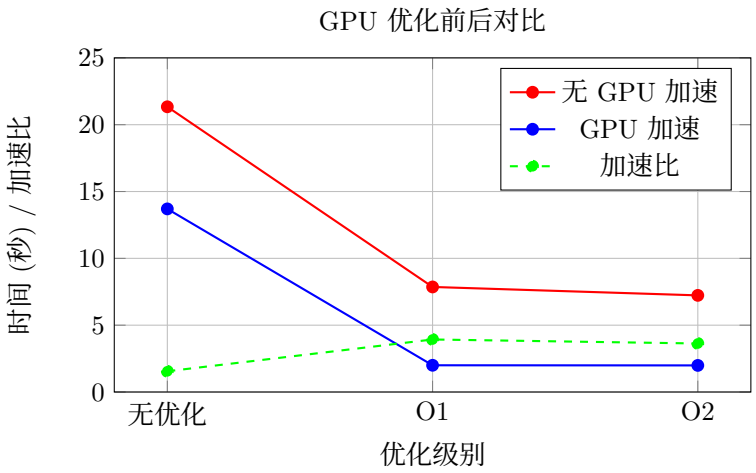
(六) GPU 加速效果

过 CUDA 并行化口令生成的关键环节，显著提高了口令生成算法的性能。为了评估 GPU 加速对口令生成的影响，我们在不同优化级别下对比了 GPU 加速和无 GPU 加速的性能差异。实验结果如下所示：

表 13: GPU 优化前后对比

优化级别	无 GPU 加速 (s)	GPU 加速 (s)	加速比
无优化	21.34	13.7	1.55
O1	7.86	2.00	3.93
O2	7.23	1.99	3.63

图 13 展示了不同优化级别下，GPU 加速与无 GPU 加速的时间对比以及加速比的变化。可以看出，GPU 加速显著缩短了口令生成的时间，尤其在高优化级别下，GPU 加速带来的性能提升更为明显。



八、 最终优化效果

融合 SIMD、多进程、MPI、GPU 加速后，对程序性能进行测试，最终结果如下：

(一) 在 ARM 服务器上的测试

该测试无法使用 GPU 优化。

表 14: ARM 服务器测试结果

优化级别	平凡算法 (s)	最终优化 (s)	加速比
无优化	21.34	14.70	1.45
O1	7.86	4.78	1.64
O2	7.23	3.98	1.81

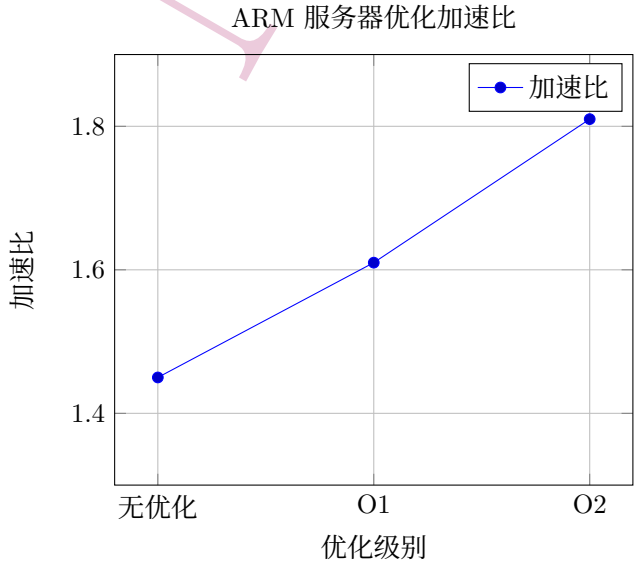


图 5: ARM 服务器优化加速比折线图

(二) 在虚拟机上的测试

具体环境已在环境配置一节中具体说明。以下是实验结果：

表 15: 虚拟机测试结果

优化级别	原始代码 (s)	最终优化 (s)	加速比
无优化	24.11	13.80	1.75
O1	8.13	2.98	2.72
O2	7.65	2.75	2.78

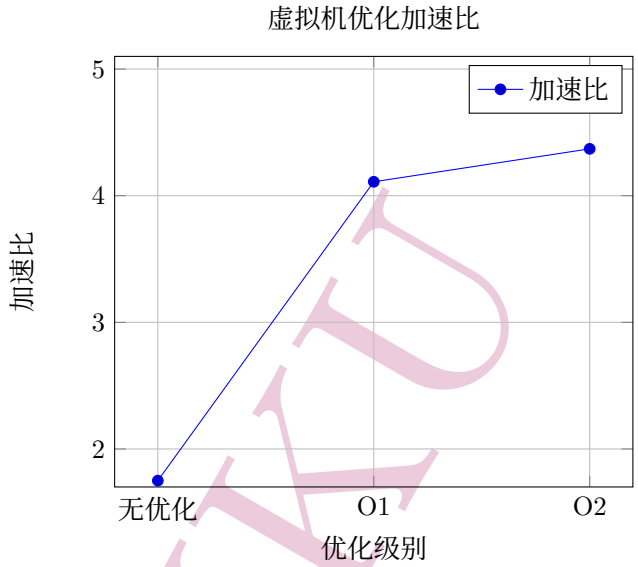


图 6: 虚拟机优化加速比折线图

可以看到，程序运行速度获得了显著提升。

(三) 总结

综上，在本学期的《并行程序设计》课程中，我深入学习并实践了包括 SIMD、多进程、MPI 以及 GPU 加速等多种并行化技术。在课程的实验过程中，我将这些并行计算方法应用于口令猜测算法，逐步优化程序的性能。通过引入 SIMD 指令集优化数据处理、使用多进程加速任务并行执行、借助 MPI 实现分布式计算，并探索 GPU 加速来进一步提升计算效率，最终使得算法获得显著的加速效果。整个过程不仅加深了我对并行编程模型的理解，也提高了我在实际开发中解决复杂性能问题的能力。