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

口令猜测算法并行化

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 4 月 29 日

目录

一、 背景

(一) 口令猜测

用户口令不是完全随机的，而是有一定语义的，遵循一定规律。我们在设置密码时，大部分情况与生日、电话号码、姓名等个人姓名相关，也与平台账号种类有关。因此猜测用户口令的基本策略是如何有效生成用户可能选择的口令和如何将生成的口令按照降序进行排列。

(二) 口令哈希算法

口令哈希算法的重要性体现在：若注册一个账号密码，并且发送给这个网站的服务器，如果此网站用明文直接存储口令，一旦服务器被攻破，口令泄露，攻击者会利用此口令登录平台账号，盗取隐私等违法行为；若此网站将口令用哈希的形式进行存储，攻破服务器也只能拿到口令的哈希值，必须用口令猜测工具尝试破解。

二、 算法原理

(一) PCFG 口令猜测

1. 串行训练

将口令按照数据类型分成不同字段：字母字段、数字字段、特殊字符。同一种口令字段按照长度分类，长度一样按照具体的 value 值分类，统计各字段频率（该猜测在密码分布中出现的概率估计，通常基于大规模真实密码泄露数据库统计每个密码或密码片段的出现频率）和 preterminal（口令中各 segment 组成的位置关系）。标记 Pivot(枢轴点): 在进行猜测变化时的修改位置索引。Pivot=0 表示从字符串的第一个字符开始修改。Pivot 最大值为字段种数，。

流程为: 初始输出猜测，系统首先生成一个猜测，以 pivot=0 位置开始修改当前猜测生成两个变化后的猜测（即只改变第一位字符）将原来的猜测的 pivot 更新递增，将修改后的 preterminal 重新插入优先队列。重复上述步骤。系统按照优先队列来管理猜测，按照概率从高到低，确保高概率猜测优先尝试，通过递增 pivot 值，系统对字符串的每个位置进行系统性修改，生成可能的变体。系统始终优先尝试统计上更可能出现的密码组合，提高破解效率。

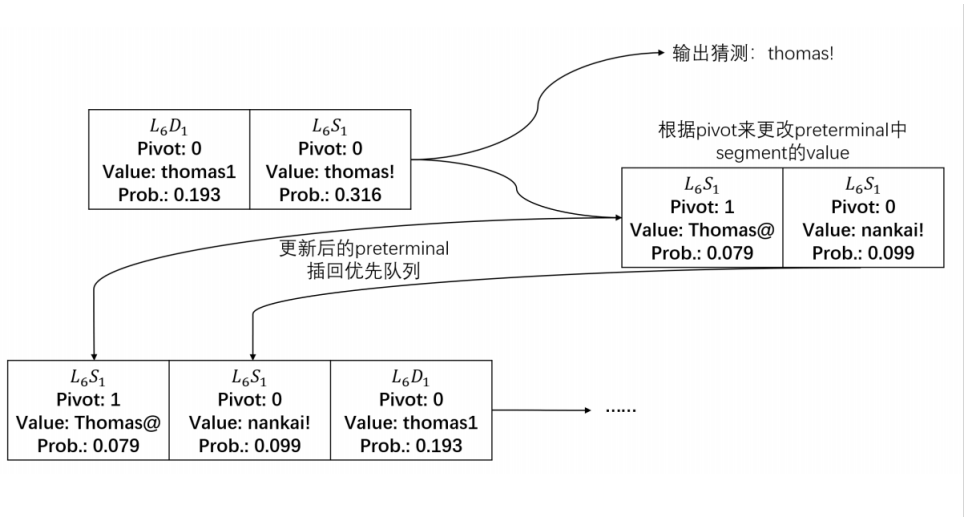

图 1: 利用优先队列进行猜测生成的过程

2. 串行猜测生成

将模型中所有的 preterminal 进行初始化，并计算各自的概率，按照概率放入优先队列中，完成优先队列的初始化。队首的一个 preterminal 出队，并且取出其中的 value 值，作为初始密码，改造（一次只改以一个字段，取这个字段对应统计数据，取出下一个概率最高的值），放回优先队列。不断重复即可不断地按概率降序生成新的口令猜测。可证明，不会生成重复口令，按概率降序，且可以遍历所有可能的排列组合。

3. 猜测生成并行化

并行化的最大难题是按照概率降序生成口令。PCFG 往往用于没有猜测次数限制的场景中（哈希破解），不需要严格降序。思路：一次性取出多个 preterminal，也可以一次为某个字段分配多个值。

流程：首先生成预密码取除最后一个标签外的所有部分并组合，然后添加修饰符，取最后一个标签的所有可能值，拼接到预密码后面。

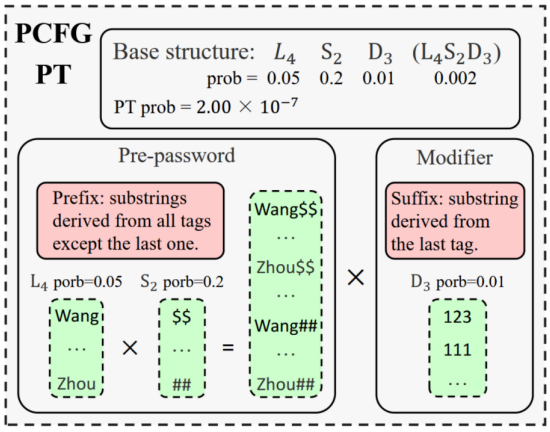

图 2: 采用的并行化方案

其他思路：采用多个优先队列并行地生成猜测；训练集分割成多个部分，并行地进行统计，最终合并。

(二) MD5 哈希算法

1. 运算过程

对消息进行预处理，变成比特串，并将其长度附加到比特串后面，新消息会被分割成多个长度为 512bit 的切片。维护一个长度为 256bit 的缓冲区，对于每个切片，分为 16 个部分，每个 32bit 的部分都会进行一轮运算，进行 16 轮运算。每次运算都会对 256bit 的缓冲区进行改变，前后依赖，顺序不可改变。所有切片都运算完成，得到缓冲区就是 MD5 的输出，也就是原始信息的哈希值。

2. 并行化思路

并行地对多个消息进行 MD5 的运算。MD5 的运算在数据上呈现出高度对齐的特性，每个数据块的处理都遵循严格的规范与顺序。并且，其运算过程具备很高的确定性，只要输入数据固

定，每次计算所得的哈希值必然一致，这使得 MD5 在数据完整性校验等场景中有着广泛且稳定的应用。

三、 代码实现

(一) guessing.cpp 中 Generate 函数

1. 内存预分配优化

```
size_t estimatedSize = pt.content.size() == 1 ?
                    pt.max_indices[0] :
                    pt.max_indices[pt.content.size() - 1];
guesses.reserve(guesses.size() + estimatedSize);
```

原理: 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。

并行化意义: 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/scatter 操作更高效，减少缓存失效，提高 ARM NEON 的加载效率。

2. 循环结构优化

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

优化细节: 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。

SIMD 适配性: 循环体简化后更容易被编译器自动向量化，向量化的关键是循环体必须足够简单且没有复杂的依赖关系；减小循环中的携带依赖，这种依赖上一次迭代结果的关系会阻碍需要同时处理多个数据的 SIMD 的并行执行；将 total_guesses 的更新操作放在线程同步的临界区中或者在所有线程完成后再统一更新，避免频繁的原子操作。

3. 分支结构优化

```
// 原始
if (type == 1) {...}
if (type == 2) {...}
if (type == 3) {...}

```

```
    // 优化后
    if (type == 1) {...}
    else if (type == 2) {...}
    else if (type == 3) {...}
```

优化效果: 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。

并行化影响: 在 SIMD 并行执行中，控制流分歧会降低并行效率，减少控制流分歧有利于 SIMD 统一执行；一旦某个条件匹配，后续的条件将不会被检查，从而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。

(二) md5.cpp 中 MD5Hash_SIMD 函数

1. 状态初始化

```
    uint32_t h0[4] = {0x67452301, 0x67452301, 0x67452301, 0x67452301};
    uint32x4_t state0 = vld1q_u32(h0);
    // 类似初始化state1-state3
```

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。

2. 输入预处理

```
    Byte* paddedMessages[4];
    int messageLengths[4];
    for (int i = 0; i < 4; i++) {
        paddedMessages[i] = StringProcess(inputs[i], &messageLengths[i]);
    }
```

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。

3. SIMD 数据加载

```
    for (int i = 0; i < 16; i++) {
        uint32_t words[4] = {0, 0, 0, 0};
        for (int j = 0; j < 4; j++) {
            if ((block + 1) * 64 <= messageLengths[j]) {
                Byte* base = paddedMessages[j] + block * 64;
                words[j] = (base[4 * i]) | (base[4 * i + 1] << 8) | ...;
            }
        }
        x[i] = vld1q_u32(words);
    }
```

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。

4. SIMD 变换函数

```
#define FF_SIMD(a, b, c, d, x, s, ac) { \
    a = vaddq_u32(a, FF_helper(b, c, d)); \
    a = vaddq_u32(a, x); \
    a = vaddq_u32(a, vdupq_n_u32(ac)); \
    a = vshlq_u32(a, vdupq_n_u32(s)); \
    a = vorrq_u32(vshrq_u32(a, vdupq_n_u32(32-s)), a); \
    a = vaddq_u32(a, b); \
}
```

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。

(三) md5.h

1. 头文件优化

添加了 include <arm_neon.h> 以支持 ARM NEON SIMD 指令，添加了头文件保护宏 ifndef MD5_H 防止重复包含。为 SIMD 指令提供必要支持提高代码健壮性。

2. 基本逻辑函数 SIMD 化

```
uint32x4_t f = vorrq_u32(vandq_u32(b, c), vandq_u32(vmvnq_u32(b), d));
```

vandq_u32：向量按位与操作，替代 &,vmvnq_u32：向量按位取反，替代,˜vorrq_u32：向量按位或操作，替代‖ 4 32

3. 循环左移 SIMD 化

```
#define ROTATELEFT_SIMD_FIXED(a, s) \
vorrq_u32(vshlq_n_u32((a), s), vshrq_n_u32((a), 32 - s))
```

vshlq_n_u32：向量左移指令，vshrq_n_u32：向量右移指令，vorrq_u32：组合左右移结果，使用立即数移位提高效率

4. 轮函数 SIMD 化

```
#define FF_SIMD(a, b, c, d, x, s, ac) { \
    uint32x4_t f = vorrq_u32(vandq_u32(b, c), vandq_u32(vmvnq_u32(b), d))
        ; \
    a = vaddq_u32(a, f); \
    a = vaddq_u32(a, x); \
```

```
        a = vaddq_u32(a, vdupq_n_u32(ac)); \
        a = ROTATELEFT_SIMD_FIXED(a, s); \
        a = vaddq_u32(a, b); \
    }
```

用 vdupq_n_u32(ac) 指令可以将一个常量广播到整个向量中，实现数据的快速复制与初始化。同时，将多个标量操作组合为单个向量操作，能够减少指令数量并优化执行流程。此外，中间结果可以保留在 SIMD 寄存器中，避免频繁的内存访问，进一步提高计算性能。

代码段 \ 优化点	串行版本	SIMD 并行版本
F 函数计算	4 次标量计算	1 次向量计算
数据加载	4 次内存访问	1 次向量加载
轮函数执行	64 次标量操作	16 次向量操作
状态更新	4 次标量加法	1 次向量加法

表 1: 优化效果对比

四、 测试结果及分析

(一) arm 平台服务器测试

编译选项 \ 优化	串行版本	SIMD 并行版本	加速比
不开编译优化	9.66795	14.8331	0.6517
编译优化 o1	2.7088	2.212	1.2246
编译优化 o2	2.78334	2.0496	1.3579

表 2: 测试哈希时间对比（单位：s）

在本实验中，使用 NEON SIMD 指令集优化了 MD5 哈希算法，并进行了不同编译选项下的性能测试。实验结果如下：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。当未加编译优化时，SIMD 并未带来预期的性能提升。而启用 -O1 或 -O2 后，明显感受到 SIMD 的并行计算带来的加速比。

1. 不开编译优化没有明显加速效果原因

NEON 函数未被正确内联：MD5_SIMD 实现依赖 NEON API,如 vaddq_u32()、vorrq_u32() 等。在 -O0 默认编译模式下，编译器不会将这些函数进行内联展开，而是将每个 SIMD 操作当作一个独立的函数调用处理。每次 SIMD 操作，程序都会产生：参数压栈, 函数跳转和返回寄存器保存。这导致每一个简单的 vaddq_u32(a, b) 都消耗了极高的调用开销，反而比直接写 C 代码的逐字节加法更慢。

循环未展开（Loop Unrolling）：MD5 算法内部有大量小循环（每一轮 16 次 step）；-O0 下，编译器逐条翻译循环体指令，不进行任何“循环展开”优化；循环体中插入了大量分支判断，造成指令流断裂、流水线停顿；SIMD 本身需要高吞吐、连续执行指令流，才能发挥性能。

无寄存器优化，频繁访存：-O0 模式下，编译器不会努力将变量留存在寄存器中；uint32x4_t 这些 SIMD 寄存器变量会频繁写回内存；访存（memory access）延迟大大增加，反而拖慢整体运行。

2. 编译优化-O1 加速原因

SIMD 指令成功内联展开：编译器在-O1 及以上级别开启了函数内联(Inlining);vaddq_u32()、vorrq_u32() 等 NEON API 直接展开为单条 ARM SIMD 指令；去掉了函数调用开销，SIMD 并行指令可以连续执行。

循环展开和向量优化：-O1 级别启用了简单循环展开，减少分支预测失败；编译器自动将连续的 SIMD 运算打包成更长的流水线执行；每个 block 的 64 字节数据被 16 次连续 SIMD 运算快速完成。

更好的寄存器调度和局部性优化：-O1/-O2 会智能分配寄存器，减少内存写回；让中间变量（比如 a, b, c, d, x[i]）一直留在寄存器里，避免访存；充分利用 CPU 的超标量执行（Superscalar Execution）能力。

3. -O2 进一步增强优化

-O2 在 -O1 基础上增加了：死代码消除（Dead Code Elimination）更加积极的循环展开（Aggressive Loop Unrolling）指令调度（Instruction Scheduling），最大限度填满 CPU 执行单元。使得 SIMD 指令能以极低的指令间隔运行，大幅提高吞吐量。

在未开启编译优化时，SIMD 指令调用开销和访存开销严重抵消了理论加速效果；而在开启 -O1 和 -O2 后，通过宏内联、循环展开、寄存器重分配和指令调度，充分释放了 NEON SIMD 的并行计算潜能，从而显著提高了 MD5 哈希处理的吞吐率。

(二) 使用 SSE 指令集在本地实现 SIMD 并行化

1. 代码

Listing 1: SSE 版本的 MD5 核心函数

```
void MD5Hash_SIMD(const string inputs[4], bit32 states[4][4]) {
    // 初始化独立状态向量，每个 lane 对应一个输入
    __m128i state0 = _mm_set1_epi32(0x67452301);
    __m128i state1 = _mm_set1_epi32(0xefcdab89);
    __m128i state2 = _mm_set1_epi32(0x98badcfe);
    __m128i state3 = _mm_set1_epi32(0x10325476);
    // 输入预处理
    Byte* paddedMessages[4];
    int messageLengths[4];
    for (int i = 0; i < 4; i++) {
        paddedMessages[i] = StringProcess(inputs[i], &messageLengths[i]);
    }
    // 计算最长输入的块数
    int maxBlocks = 0;
    for (int i = 0; i < 4; ++i)
        maxBlocks = std::max(maxBlocks, messageLengths[i] / 64);
    // 主循环，逐 block 处理
```

```
        for (int block = 0; block < maxBlocks; block++) {
            __m128i x[16];
            for (int i = 0; i < 16; i++) {
                uint32_t words[4] = {0, 0, 0, 0};
                for (int j = 0; j < 4; j++) {
                    if ((block + 1) * 64 <= messageLengths[j]) {
                        Byte* base = paddedMessages[j] + block * 64;
                        words[j] = (base[4 * i]) |
                                   (base[4 * i + 1] << 8) |
                                   (base[4 * i + 2] << 16) |
                                   (base[4 * i + 3] << 24);
                    }
                }
                x[i] = _mm_loadu_si128((__m128i*)words);
            }
            __m128i a = state0, b = state1, c = state2, d = state3;
            // Round 1
            FF_SIMD(a, b, c, d, x[0], s11, 0xd76aa478);
            FF_SIMD(d, a, b, c, x[1], s12, 0xe8c7b756);
            FF_SIMD(c, d, a, b, x[2], s13, 0x242070db);
            FF_SIMD(b, c, d, a, x[3], s14, 0xc1bdceee);
            FF_SIMD(a, b, c, d, x[4], s11, 0xf57c0faf);
            FF_SIMD(d, a, b, c, x[5], s12, 0x4787c62a);
            FF_SIMD(c, d, a, b, x[6], s13, 0xa8304613);
            FF_SIMD(b, c, d, a, x[7], s14, 0xfd469501);
            FF_SIMD(a, b, c, d, x[8], s11, 0x698098d8);
            FF_SIMD(d, a, b, c, x[9], s12, 0x8b44f7af);
            FF_SIMD(c, d, a, b, x[10], s13, 0xffff5bb1);
            FF_SIMD(b, c, d, a, x[11], s14, 0x895cd7be);
            FF_SIMD(a, b, c, d, x[12], s11, 0x6b901122);
            FF_SIMD(d, a, b, c, x[13], s12, 0xfd987193);
            FF_SIMD(c, d, a, b, x[14], s13, 0xa679438e);
            FF_SIMD(b, c, d, a, x[15], s14, 0x49b40821);

            // Round 2~4省略

            state0 = _mm_add_epi32(state0, a);
            state1 = _mm_add_epi32(state1, b);
            state2 = _mm_add_epi32(state2, c);
            state3 = _mm_add_epi32(state3, d);
        }
        // 将 SSE state 存回到输出数组
        uint32_t result[4];
        _mm_storeu_si128((__m128i*)result, state0);
        for (int i = 0; i < 4; i++) states[i][0] = __builtin_bswap32(result[i
            ]);
        _mm_storeu_si128((__m128i*)result, state1);
        for (int i = 0; i < 4; i++) states[i][1] = __builtin_bswap32(result[i
```

```
            ]);
65          _mm_storeu_si128((__m128i*)result, state2);
66          for (int i = 0; i < 4; i++) states[i][2] = __builtin_bswap32(result[i
                ]);
67          _mm_storeu_si128((__m128i*)result, state3);
68          for (int i = 0; i < 4; i++) states[i][3] = __builtin_bswap32(result[i
                ]);
69          // 清理内存
70          for (int i = 0; i < 4; i++) delete[] paddedMessages[i];
71      }
```

将 uint32x4_t 替换为 __m128i 将 vld1q_u32() 替换为 _mm_loadu_si128()
将 vst1q_u32() 替换为 _mm_storeu_si128()
将 vaddq_u32() 替换为 _mm_add_epi32()
使用 _mm_set1_epi32() 进行初始化，替代 NEON 的实现方式

Listing 2: SSE 版本的 md5.hframe

```
1   #include <iostream>
2   #include <string>
3   #include <cstring>
4   #include <emmintrin.h>  // SSE2
5   #include <tmmintrin.h>  // SSSE3
6
7   using namespace std;
8
9   typedef unsigned char Byte;
10  typedef unsigned int bit32;
11
12  // 移位常量省略
13
14  // 普通 MD5 函数
15  #define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
16  #define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
17  #define H(x, y, z) ((x) ^ (y) ^ (z))
18  #define I(x, y, z) ((y) ^ ((x) | (~z)))
19
20  #define ROTATELEFT(num, n) (((num) << (n)) | ((num) >> (32-(n))))
21
22  #define FF(a, b, c, d, x, s, ac) { \
23    (a) += F ((b), (c), (d)) + (x) + ac; \
24    (a) = ROTATELEFT ((a), (s)); \
25    (a) += (b); \
26  }
27
28  #define GG(a, b, c, d, x, s, ac) { \
29    (a) += G ((b), (c), (d)) + (x) + ac; \
30    (a) = ROTATELEFT ((a), (s)); \
31    (a) += (b); \
```

```
}

#define HH(a, b, c, d, x, s, ac) { \
  (a) += H ((b), (c), (d)) + (x) + ac; \
  (a) = ROTATELEFT ((a), (s)); \
  (a) += (b); \
}

#define II(a, b, c, d, x, s, ac) { \
  (a) += I ((b), (c), (d)) + (x) + ac; \
  (a) = ROTATELEFT ((a), (s)); \
  (a) += (b); \
}

// SIMD RotateLeft（立即数版本）
#define ROTATELEFT_SIMD_FIXED(a, s) \
    _mm_or_si128(_mm_slli_epi32((a), s), _mm_srli_epi32((a), 32 - s))

// SIMD 宏（使用立即数移位，编译器完全支持）
#define FF_SIMD(a, b, c, d, x, s, ac) { \
    __m128i f = _mm_or_si128(_mm_and_si128(b, c), _mm_and_si128(
        _mm_andnot_si128(b, _mm_set1_epi32(-1)), d)); \
    a = _mm_add_epi32(a, f); \
    a = _mm_add_epi32(a, x); \
    a = _mm_add_epi32(a, _mm_set1_epi32(ac)); \
    a = ROTATELEFT_SIMD_FIXED(a, s); \
    a = _mm_add_epi32(a, b); \
}

//省略

// 接口声明
void MD5Hash(string input, bit32 *state);
void MD5Hash_SIMD(const string inputs[4], bit32 states[4][4]);
```

vandq_u32 → _mm_and_si128
vorrq_u32 → _mm_or_si128
vmvnq_u32 → _mm_andnot_si128 (with _mm_set1_epi32(-1))
veorq_u32 → _mm_xor_si128
vaddq_u32 → _mm_add_epi32
vshlq_n_u32 → _mm_slli_epi32
vshrq_n_u32 → _mm_srli_epi32
vdupq_n_u32 → _mm_set1_epi32

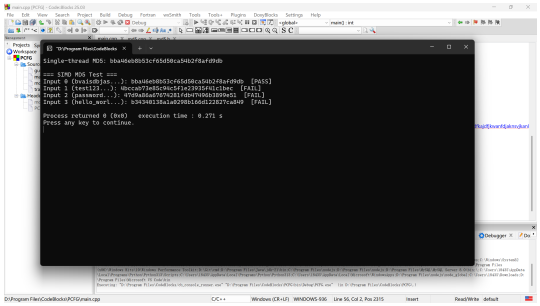

图 3: 验证正确性

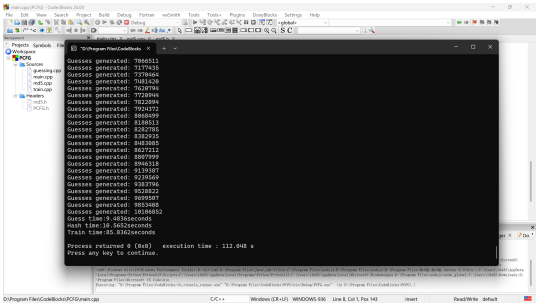

图 4: 采用的并行化方案

图 5: 实验结果

2. perf 分析串行和并行

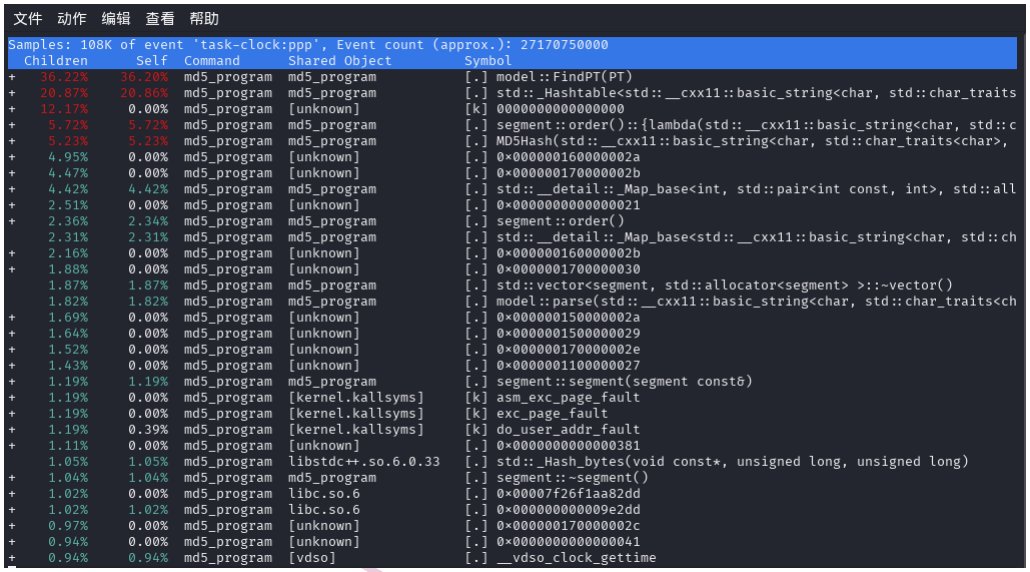
文件 动作 编辑 查看 帮助
Samples: 108K of event 'task-clock:ppp', Event count (approx.): 27170750000
Children Self Command Shared Object Symbol
+ 36.22% 36.20% md5_program md5_program [.] model::FindPT(PT)
+ 20.87% 20.86% md5_program md5_program [.] std::_Hashtable<std::__cxx11::basic_string<char, std::char_traits
+ 12.17% 0.00% md5_program [unknown] [k] 0000000000000000
+ 5.72% 5.72% md5_program md5_program [.] segment::order()::{lambda(std::__cxx11::basic_string<char, std::c
+ 5.23% 5.23% md5_program md5_program [.] MD5Hash(std::__cxx11::basic_string<char, std::char_traits<char>,
+ 4.95% 0.00% md5_program [unknown] [.] 0x000000160000002a
+ 4.47% 0.00% md5_program [unknown] [.] 0x000000170000002b
+ 4.42% 4.42% md5_program md5_program [.] std::__detail::_Map_base<int, std::pair<int const, int>, std::all
+ 2.51% 0.00% md5_program [unknown] [.] 0x0000000000000021
+ 2.36% 2.34% md5_program md5_program [.] segment::order()
2.31% 2.31% md5_program md5_program [.] std::__detail::_Map_base<std::__cxx11::basic_string<char, std::ch
+ 2.16% 0.00% md5_program [unknown] [.] 0x000000160000002b
+ 1.88% 0.00% md5_program [unknown] [.] 0x0000001700000030
1.87% 1.87% md5_program md5_program [.] std::vector<segment, std::allocator >::~vector()
1.82% 1.82% md5_program md5_program [.] model::parse(std::__cxx11::basic_string<char, std::char_traits<ch
+ 1.69% 0.00% md5_program [unknown] [.] 0x000000150000002a
+ 1.64% 0.00% md5_program [unknown] [.] 0x0000001500000029
+ 1.52% 0.00% md5_program [unknown] [.] 0x000000170000002e
+ 1.43% 0.00% md5_program [unknown] [.] 0x0000001100000027
+ 1.19% 1.19% md5_program md5_program [.] segment::segment(segment const&)
+ 1.19% 0.00% md5_program [kernel.kallsyms] [k] asm_exc_page_fault
+ 1.19% 0.00% md5_program [kernel.kallsyms] [k] exc_page_fault
+ 1.19% 0.39% md5_program [kernel.kallsyms] [k] do_user_addr_fault
+ 1.11% 0.00% md5_program [unknown] [.] 0x0000000000000381
1.05% 1.05% md5_program libstdc++.so.6.0.33 [.] std::_Hash_bytes(void const*, unsigned long, unsigned long)
+ 1.04% 1.04% md5_program md5_program [.] segment::~segment()
+ 1.02% 0.00% md5_program libc.so.6 [.] 0x00007f26f1aa82dd
+ 1.02% 1.02% md5_program libc.so.6 [.] 0x000000000009e2dd
+ 0.97% 0.00% md5_program [unknown] [.] 0x000000170000002c
+ 0.94% 0.00% md5_program [unknown] [.] 0x0000000000000041
+ 0.94% 0.94% md5_program [vdso] [.] __vdso_clock_gettime

图 6: 串行性能

函数	Self 占比	分析
`FindPT`	36.20%	非常高！这可能是你自己定义的主循环或者消息调度函数；它占了**超过 1/3 的总时间**。极有可能是哈希主循环或数据准备部分。
`std::_Hashtable<std::string,…>`	20.86%	使用了**哈希表（如 unordered_map）处理字符串**，代价很高！说明你在 unordered_map 中频繁插入/查找字符串，严重拖累性能。
`segment::order()` / `Hash()` / `operator==`	∼12%	表明你可能用 segment 作为 unordered_map 的 key，还实现了自定义哈希函数和比较操作，这些操作非常频繁地调用。
`_ZNSt6vectorISegmentSaIS0_EEEC2Ev`	2.3%	频繁构造 `vector<segment>`，可能是你在处理输入或中间状态时不断创建新的对象（建议复用）。
`Hash_bytes(void const*, unsigned long, unsigned long)`	1.05%	这是**真正的哈希函数核心**（比如 MD5），只占了 1%，说明 MD5 本身**不是瓶颈**。

表 3: perf 分析中的热点函数及其占用情况

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

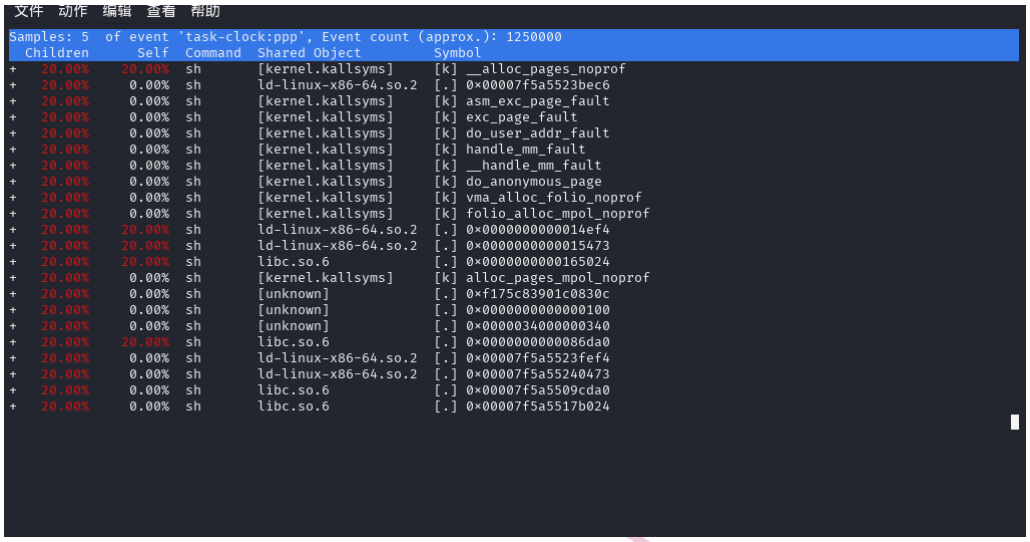

图 7: 并行性能

模块	Self 占比	说明
`__alloc_pages_noprof`	20.00%	低层次的物理页分配函数，分配匿名页（匿名内存）。
`do_anonymous_page`	20.00%	处理没有映射文件的页面（比如 `malloc()` 分配的大块内存）。
`handle_mm_fault` / `exc_page_fault` / `asm_exc_page_fault`	20.00%	访问未映射地址时触发，属于懒分配（page fault）。
`libc.so.6` 匿名函数	20.00%	libc 内部操作，很可能是系统调用包装层（如 `malloc`、线程管理）。
`ld-linux-x86-64.so.2` 匿名函数	20.00%	动态链接器相关开销（动态加载共享库）。

表 4: perf 分析中并行版本的热点模块及其说明

3. SIMD 计算两个、四个、八个消息性能分析

并行度	理论吞吐量	实际影响	可能现象
2	每次算 2 条消息	SIMD 指令利用率低，吞吐下降，效率降低	性能差，约下降 40%-50%
4	每次算 4 条消息	最适合 NEON 宽度，硬件完美支持	最佳性能
8	理论上更高，但要特殊硬件	如果搞定，吞吐会接近翻倍	需要更高级指令集，否则不可行

表 5: 不同并行度下的吞吐量与性能变化

五、 总结与收获

(一) 总结

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。

(二) 收获

- 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。
- 实践了 ARM NEON 和 x86 SSE 指令集的使用，掌握了从普通 C++ 循环到向量化指令改写的具体技巧。
- 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。
- 收获了工程层面对大规模数据处理程序进行并行重构与性能调优的完整流程经验。

(三) Github 网址

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