南開大學
Nankai University

计算机学院

并行程序设计报告

并行加速的口令猜测算法

姓名：潘涛

学号：2314033

专业：计算机科学与技术

2025 年 4 月 29 日

目录

1 实验环境

本次实验将对 MD5 哈希算法进行 SIMD 并行加速，实验相关平台及环境如下表所示。

实验平台	ARM 平台（华为鲲鹏服务器）
Linux 内核版本	5.10.0-235.0.0.134.oe2203sp4.aarch64
编译器	GCC
SIMD 指令集	NEON
profiling	perf

2 问题描述

对于一个用户的口令，对其进行猜测的基本策略是：生成一个按照概率降序排列的口令猜测词典，这个词典包括一系列用户可能选择的口令。那么，现在问题就变成了：1. 如何有效生成用户可能选择的口令；2. 如何将生成的口令按照降序进行排列。在这个选题中，使用最经典的 PCFG（Probabilistic Context-Free Grammar，概率上下文无关文法）模型来进行口令的生成，并且尝试将其并行化，以提升猜测的时间效率。口令猜测选题的框架主要分为三个部分：模型训练、口令生成、MD5 哈希值生成。在本篇中，我们将着重讨论 MD5 哈希算法的 SIMD 并行实现。

MD5 是一个常用的哈希算法。对于任意长度的信息（字符串，文件，图像等），MD5 都能为其生成一个固定长度的 “摘要”。给定一个消息，MD5 将能够生成一个确定性的 “摘要”，并且对原始消息的任意改动都会改变 MD5 的哈希结果。不同的消息通过 MD5 产生相同输出的概率是极低的。

MD5 算法大致的运算过程如下。MD5 首先将对消息进行预处理，将其变成比特串的形式，并将其长度附加到这个比特串的后面。然后，这个附加了消息长度的新消息会被分割成多个长度为 512bit 的切片。对于不足 512bit 的部分，将会填充值 512bit。然后，我们将会维护一个长度为 256bit 的缓冲区。对于每一个长度为 512bit 的切片，我们会将其分为长度为 32bit 的 16 个部分。每个 32bit 都会分别用于参与一轮运算，也就是说每个长度为 512bit 的切片将会经过 16 轮运算。每轮运算使用的计算公式不尽相同，但每次运算都会对 256bit 的缓冲区进行改变，并且这 16 轮运算是前后依赖的，顺序不可改变。最终，当所有 512bit 切片都运算完毕后，得到的最终 256bit 缓冲区就是 MD5 的输出，也就是原始消息的哈希值。原理图如下：

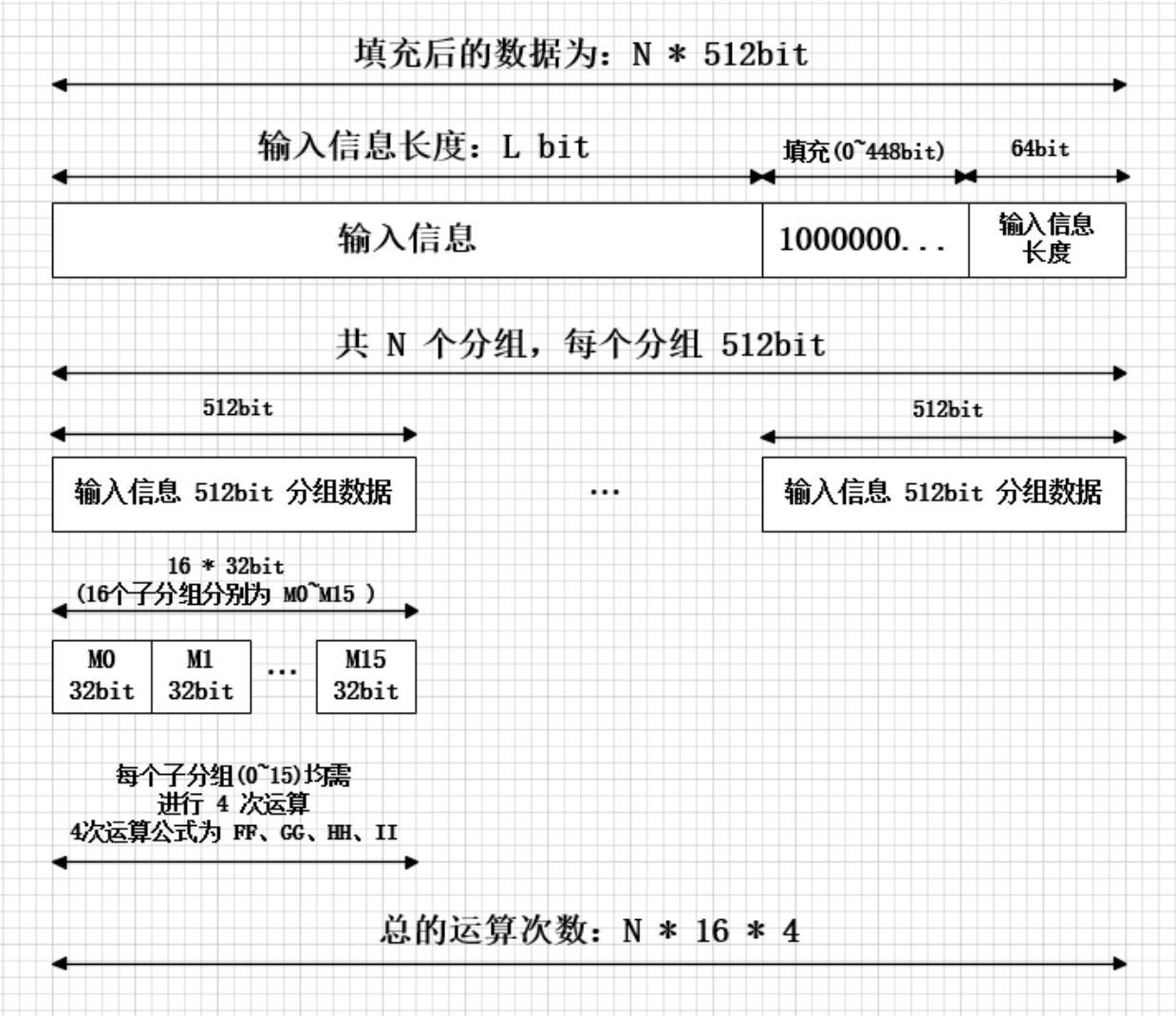
3 算法设计

3.1 串行算法

3.1.1 整体实现

对于串行算法，实现函数 MD5Hash，每次将处理一个口令（密码）的 hash 值计算。对于算法实现，MD5Hash 函数接收到一个指令 input。然后使用 MD5 预处理的步骤，通过 StringProcess 函数对输入字符串进行填充，使其长度符合 MD5 的要求，结果保存在 paddedMessage 中，并记录了处理后的总长度到 messageLength[0] 中。然后，将会对数据进行切分为多个 512bit 的切片，并使用 n_block 记录切片数量。对于每个 512bit 的切片，将会被进一步切分 16 个 32bit 的部分保存在下 x[i] 中。对于每个小块的长度是 32bit（4Byte），即每个 x[i] 的值是数组 paddedMessage 的连续 4 个数据。接着是执行 4 轮 MD5 压缩变换（FF, GG, HH, II），每个回合使用不同的非线性函数（F/G/H/I）和不同常量，将消息 x 与当前变量 a, b, c, d 反复混合、旋转、累加。每一回合会调用 16 次对应的宏函数。

将本轮变换得到的结果加到全局状态 state 上，这是 MD5 的链式结构精髓。最后将缓冲区 state 中的值从小端格式转为大端格式，方便输出或比较。



串行 MD5 哈希算法

```
void MD5Hash(string input, bit32 *state)
{
  Byte *paddedMessage;
  int *messageLength = new int[1];
  for (int i = 0; i < 1; i += 1)
  {
    paddedMessage = StringProcess(input, &messageLength[i]);
    assert(messageLength[i] == messageLength[0]);
  }
  int n_blocks = messageLength[0] / 64;

  state[0] = 0x67452301;
  state[1] = 0xefcdab89;
  state[2] = 0x98badcfe;
```

```
15    state[3] = 0x10325476;
16
17    // 逐block地更新state
18    for (int i = 0; i < n_blocks; i += 1)
19    {
20      bit32 x[16];
21
22      // 下面的处理，在理解上较为复杂
23      for (int i1 = 0; i1 < 16; ++i1)
24      {
25        x[i1] = (paddedMessage[4 * i1 + i * 64]) |
26            (paddedMessage[4 * i1 + 1 + i * 64] << 8) |
27            (paddedMessage[4 * i1 + 2 + i * 64] << 16) |
28            (paddedMessage[4 * i1 + 3 + i * 64] << 24);
29      }
30
31      bit32 a = state[0], b = state[1], c = state[2], d = state[3];
32
33      auto start = system_clock::now();
34      // Round 1
35      FF(a, b, c, d, x[0], s11, 0xd76aa478);
36      ......
37      GG(a, b, c, d, x[1], s21, 0xf61e2562);
38      ......
39      HH(a, b, c, d, x[5], s31, 0xfffa3942);
40      ......
41      II(a, b, c, d, x[0], s41, 0xf4292244);
42
43      state[0] += a;
44      state[1] += b;
45      state[2] += c;
46      state[3] += d;
47    }
48    for (int i = 0; i < 4; i++)
49    {
50      uint32_t value = state[i];
51      state[i] = ((value & 0xff) << 24) |     // 将最低字节移到最高位
52            ((value & 0xff00) << 8) |    // 将次低字节左移
53            ((value & 0xff0000) >> 8) |    // 将次高字节右移
54            ((value & 0xff000000) >> 24); // 将最高字节移到最低位
55    }
56    // 释放动态分配的内存
57    delete[] paddedMessage;
58    delete[] messageLength;
59  }
```

3.1.2 具体函数方法实现

预处理函数 StringProcess 的实现。主要完成以下三个关键步骤：首先，计算原始消息长度并确定填充位数（paddingBits），通过取模运算确保填充后消息长度满足 512 比特的整数倍减去 64 比特（即 448 比特模 512）；其次，进行实际填充操作，在消息末尾添加一个 0x80 字节和若干 0x00 字节，使消息长度达到 448 比特的边界；最后，追加 64 比特的原始消息长度值（小端序存储），从而构造出符合 MD5 算法要求的 512 比特整数倍长度的输入消息块。该实现通过动态内存分配创建填充后的消息缓冲区，使用 memcpy 复制原始数据，并采用位操作高效处理长度值的字节序转换，最终输出填充后的消息指针及其长度，为后续的 MD5 哈希计算提供标准化的输入格式。

串行 MD5 哈希算法相关函数定义

```
Byte *StringProcess(string input, int *n_byte)
{
  Byte *blocks = (Byte *)input.c_str();
  int length = input.length();
  int bitLength = length * 8;
  int paddingBits = bitLength % 512;
  if (paddingBits > 448)
  {
    paddingBits = 512 - (paddingBits - 448);
  }
  else if (paddingBits < 448)
  {
    paddingBits = 448 - paddingBits;
  }
  else if (paddingBits == 448)
  {
    paddingBits = 512;
  }
  int paddingBytes = paddingBits / 8;
  int paddedLength = length + paddingBytes + 8;
  Byte *paddedMessage = new Byte[paddedLength];
  memcpy(paddedMessage, blocks, length);
  paddedMessage[length] = 0x80;                // 添加一个0x80字节
  memset(paddedMessage + length + 1, 0, paddingBytes - 1); // 填充0字节

  for (int i = 0; i < 8; ++i)
  {
    paddedMessage[length + paddingBytes + i] = ((uint64_t)length * 8 >> (i * 8)) &
        0xFF;
  }

  // 验证长度是否满足要求。此时长度应当是512bit的倍数
  int residual = 8 * paddedLength % 512;

  *n_byte = paddedLength;
  return paddedMessage;
}
```

我们可以看到所有支持 MD5Hash 函数的支持函数都是使用 #define 定义，这是由于 #define 是预处理指令，它不会在运行时执行，而是在编译前进行简单的文本替换，所以不需要“返回值”，而是在使用改指令时，直接替换为相关的表达式。使用宏定义，而不是函数实现，则是因为实现的 F、G、H、I、HH 等操作都是简单位运算，而宏定义是直接代码展开，不存在函数调用的过程，不占用额外栈内存，可以减小调用的开销，从而加快程序执行的效率。观察这些函数，这些函数所设计的运算不涉及条件判断，并且为较简单的运算（移位、与运算），这就使得这些函数非常适合用 SIMD 进行并行化处理：一次性地让函数同时处理多个消息（口令），即可做到并行化。后续的 SIMD 并行优化，就是主要针对这些函数进行优化，实现向量化数据的计算。

串行 MD5 哈希算法相关函数定义

```
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))

#define ROTATELEFT(num, n) (((num) << (n)) | ((num) >> (32-(n))))

#define FF(a, b, c, d, x, s, ac) { \
  (a) += F ((b), (c), (d)) + (x) + ac; \
  (a) = ROTATELEFT ((a), (s)); \
  (a) += (b); \
}
#define GG(a, b, c, d, x, s, ac) { \
  (a) += G ((b), (c), (d)) + (x) + ac; \
  (a) = ROTATELEFT ((a), (s)); \
  (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); \
}
```

3.2 SIMD 并行化算法

对于 MD5 哈希算法的优化和加速，我们将采用 SIMD 实现。SIMD（Single Instruction, Multiple Data，单指令多数据）是一种并行计算技术，它允许处理器在一个指令周期内同时对多个数据执行相同的操作。通过将多个数据打包成向量并并行处理，SIMD 能显著提高处理图像、音频、视频、科学计算和机器学习等场景中的数据处理效率。常见的 SIMD 指令集包括 Intel 的 SSE/AVX 和 ARM 的 NEON。在本实验中，我们将在 ARM 平台使用 NEON 指令集实现 MD5 哈希算法的并行化。串行的 MD5 哈希算法一次计算一个口令的 hash 值，而我们 SIMD 并行的主要思路是一次性地让函数同时处

理 4 个口令的 hash 值。

3.2.1 详细实现

我们使用同时输入四个口令保持在 inputs 中，对于缓冲区 states, 我们使用 neon 指令集中的数据类型 uint32x4 替代，即每个 states 相当于包含 4 个无符号 32 位整数的向量，这样实现了同时 4 个口令的缓冲区。然后我们利用 StringProcess 对 4 个口令依次进行预处理，并保存在 paddedMessages 中。接着是对于四个 128bit 的缓存区赋予初值，使用 neon 中的 vdupq_n_u32 会将 32-bit 整数复制到一个 128-bit 向量 state0 中的每一个元素。

并行 MD5 哈希算法

```
void MD5Hash_new(string inputs[4], uint32x4_t states[4])
{
  Byte *paddedMessages[4];
    int messageLengths[4];
  for (int i = 0; i < 4; i++)
  {
    paddedMessages[i] = StringProcess(inputs[i], &messageLengths[i]);
        if (i > 0 && messageLengths[i] != messageLengths[0]) {
            messageLengths[i] = messageLengths[0];
        }
  }


    uint32x4_t state0 = vdupq_n_u32(0x67452301);
    uint32x4_t state1 = vdupq_n_u32(0xefcdab89);
    uint32x4_t state2 = vdupq_n_u32(0x98badcfe);
    uint32x4_t state3 = vdupq_n_u32(0x10325476);
    ......
```

接下来是逐个 block 的更新缓冲区 states。这段代码实现了将 4 条消息的当前块中第 i 个 32 位字提取出来，并打包成一个 128 位的 SIMD 向量，用于并行处理。它通过逐条消息读取对应的 4 个字节，按小端格式拼接成一个 32 位整数，最终用 NEON 指令 vld1q_u32 将 4 个整数加载到一个向量寄存器中，供后续 MD5 并行计算使用。接着是 16 轮的计算保持不变。最后更新缓存区，使用指令 vaddq_u32 将两个 128 位的向量 states 和 a 中对应的 4 个 32 位无符号整数进行逐元素相加，并返回一个新的向量结果。

并行 MD5 哈希算法

```
    // 逐个block地更新states
  for (int block = 0; block < n_blocks; block ++)
  {
    uint32x4_t a = state0, b = state1, c = state2, d = state3;
        uint32x4_t x[16];
        for(int i = 0; i < 16; i++) {
            uint32_t words[4];
            for(int j = 0; j < 4; j++) {
                const Byte* blockStart = paddedMessages[j] + block * 64;
```

```
                words[j] = (blockStart[i*4]) |
                           (blockStart[i*4 + 1] << 8) |
                           (blockStart[i*4 + 2] << 16) |
                           (blockStart[i*4 + 3] << 24);
            }
            x[i] = vld1q_u32(words);
        }
    //同样的四轮计算保持不变
        ......

    // 更新状态
    states[0] = vaddq_u32(states[0], a);
    states[1] = vaddq_u32(states[1], b);
    states[2] = vaddq_u32(states[2], c);
    states[3] = vaddq_u32(states[3], d);
}
```

这段代码通过 vreinterpretq_u8_u32 和 vrev32q_u8 等 NEON 指令，对 MD5 哈希结果的每个 32 位整数进行字节序转换。首先，vreinterpretq_u8_u32 将四个 uint32_t 组成的向量 states[i] 重新解释为 16 字节的 uint8_t 向量；接着，vrev32q_u8 对每组 4 字节进行字节逆序（低位字节与高位字节交换）；最后，再通过 vreinterpretq_u32_u8 将结果转回 uint32x4_t 类型，实现小端到大端的转换，确保输出顺序符合 MD5 规范。

并行 MD5 哈希算法

```
    states[0] = vreinterpretq_u32_u8(vrev32q_u8(vreinterpretq_u8_u32(states[0])));
    states[1] = vreinterpretq_u32_u8(vrev32q_u8(vreinterpretq_u8_u32(states[1])));
    states[2] = vreinterpretq_u32_u8(vrev32q_u8(vreinterpretq_u8_u32(states[2])));
    states[3] = vreinterpretq_u32_u8(vrev32q_u8(vreinterpretq_u8_u32(states[3])));
```

下面的代码我们将利用 neon 指令集中对应的针对 4 个 32 位向量的计算指令，实现 F、G、H、I、FF、GG、HH、II 的并行实现。

并行 MD5 哈希算法

```
#define F_NEON(x, y, z) vorrq_u32(vandq_u32(x, y), vandq_u32(vmvnq_u32(x), z))
#define G_NEON(x, y, z) vorrq_u32(vandq_u32(x, z), vandq_u32(vmvnq_u32(z), y))
#define H_NEON(x, y, z) veorq_u32(veorq_u32(x, y), z)
#define I_NEON(x, y, z) veorq_u32(y, vorrq_u32(x, vmvnq_u32(z)))

#define ROTATELEFT_NEON(x, n) vorrq_u32(vshlq_n_u32(x, n), vshrq_n_u32(x, 32 - n))

#define FF(a, b, c, d, x, s, ac) { \
    uint32x4_t t = F_NEON(b, c, d); \
    a = vaddq_u32(a, t); \
    a = vaddq_u32(a, x); \
    a = vaddq_u32(a, vdupq_n_u32(ac)); \
    a = ROTATELEFT_NEON(a, s); \
    a = vaddq_u32(a, b); \
}
```

```
//GG,HH, I I 实 现 类 似
```

代码中利用的相关 neon 指令及其含义如下表：

vorrq_u32(a,b)	对两个 128 位向量 a 和 b 中的每个对应 32 位无符号整数元素执行按位“或”操作
vandq_u32(a,b)	对两个 128 位向量 a 和 b 中每个对应的 32 位无符号整数执行按位与操作
vmvnq_u32(x)	对一个 128 位向量 a 中的每个 32 位无符号整数执行按位取反操作
veorq_u32(a,b)	对两个 128 位向量 a 和 b 中每个对应的 32 位无符号整数执行按位异或操作
vaddq_u32(a,b)	对于两个 128 位向量 a,b 中中每 32 位相加

我们更改了 MD5 哈希算法的输入，现在一些将会同时输入 4 个口令，因此在口令猜测的主程序中我们也要修改对应部分的代码。具体的实现如下，依次取出口令，每 4 个为一组，不足 4 个空字符串填充。

并行 MD5 哈希算法

```
auto it = q.guesses.begin();
while (it != q.guesses.end()) {
    // 批量填充4个密码
    string inputs[4];
    int batch_count = 0;
    while (it != q.guesses.end() && batch_count < 4) {
        inputs[batch_count++] = *it++;
    }

    // 不足4个时填充空字符串
    for (int i = batch_count; i < 4; ++i) {
        inputs[i].clear();
    }

    uint32x4_t states[4];
    MD5Hash_new(inputs, states);
}
```

4 性能测试

4.1 正确性检测

为了检验 SIMD 实现的 MD5 哈希算法的正确性，我们将通过两个算法针对同一个字符串的哈希值进行比较。在 correctness.cpp 中，我们将字串”” 作为函数输入。

```
[s2314033@master_ubss1 guess]$ g++ correctness.cpp train.cpp guessing.cpp md5.cpp -o test.exe
[s2314033@master_ubss1 guess]$ ./test.exe
bba46eb8b53cf65d50ca54b2f8afd9db
```

使用串行得到的结果是：bba46eb8b53cf65d50ca54b2f8afd9db。

使用 SIMD 得到的结果是：bba46eb8b53cf65d50ca54b2f8afd9db。

两个算法得到得结果相同，说明我们实现得 MD5 哈希算法是正确的，，能够正确的输出哈希值。

4.2 测试结果

为了评估串行实现和 SIMD 并行实现的性能表现，我们将针对口令猜测程序的运行时间进行计时，然后利用 hashtime 来比较两个算法的性能表现。对于程序的编译，我们同一采用指令:g++ main.cpp train.cpp guessing.cpp md5.cpp -o main -O1，即使用 O1 优化实现编译来测试性能表现。测试结果如表2所示。

	hash time	guess time	train time
串行算法	3.159	0.623	29.336
SIMD 实现	2.086	0.647	26.688

表 1: 性能测试结果 (o1 优化)(单位:s)

对于本次实验我们着重在于 MD5 哈希算法的优化，所以我们更加关注 hash time，从表中可以看到 SIMD 并行化后实现的 MD5 哈希算法相对于串行实现了加速，加速比为 1.514。

5 profiling

我们将利用 perf 工具来测试程序的性能表现，依这分析不同表现的原因。使用命令 perf stat ./main 用来快速统计程序整体性能指标. 使用命令 perf record ./main perf report 查看具体的函数占用。

5.1 串行算法

对于串行算法，输出结果如下图；

```
         36,964.37 msec task-clock:u              #    0.998 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
            88,522      page-faults:u             #    0.002 M/sec
    93,848,130,390      cycles:u                  #    2.539 GHz
    60,296,785,551      instructions:u            #    0.64  insn per cycle
  <not supported>      branches:u
       646,881,300      branch-misses:u

      37.042371127 seconds time elapsed

      36.361173000 seconds user
       0.470789000 seconds sys
```

图 5.1: 串行

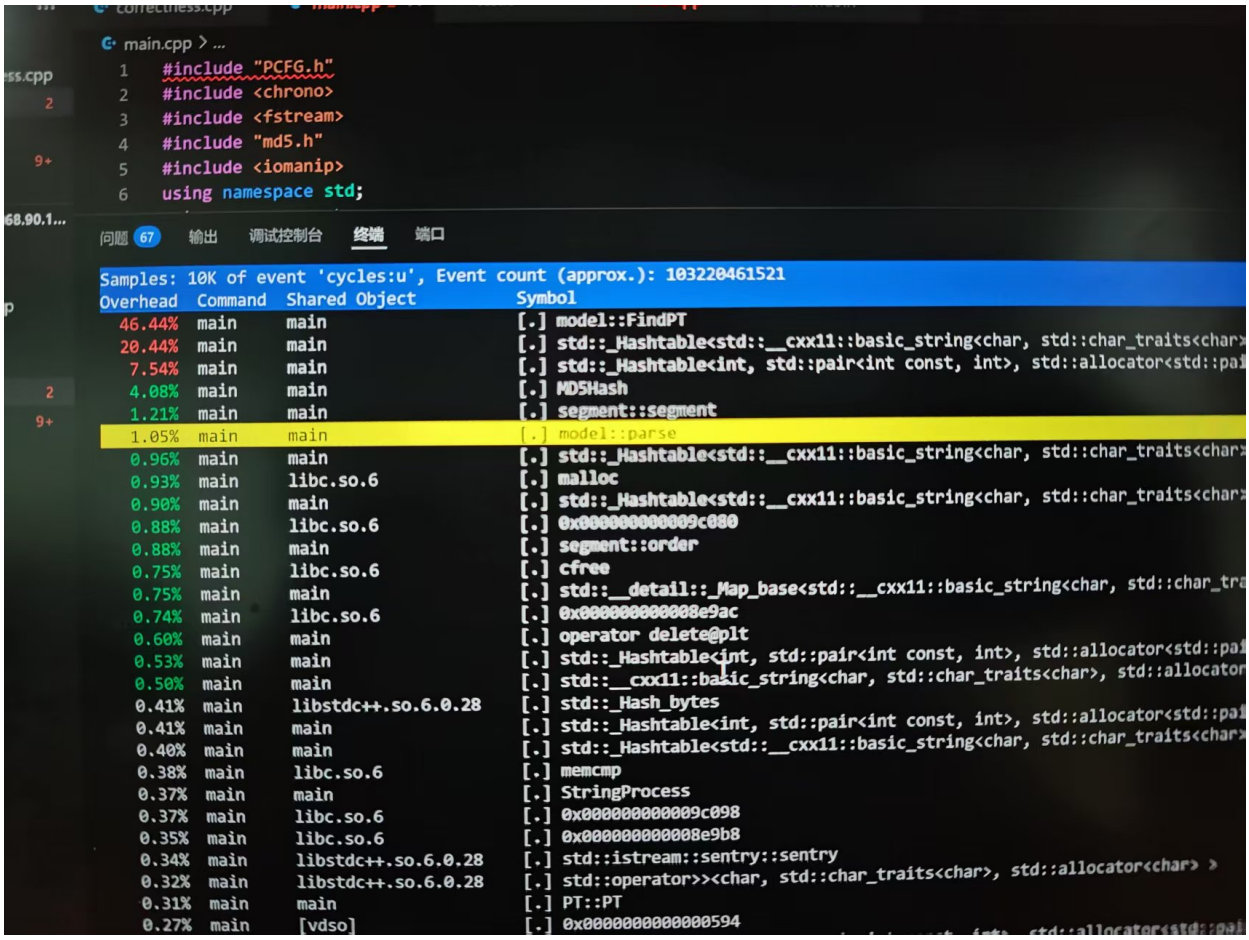


图 5.2: 串行

5.2 SIMD 并行

对于采用 SIMD 实现的并行后，测试结果如下：

```
     37,205.87 msec task-clock:u              #    1.000 CPUs utilized
             0      context-switches:u        #    0.000 K/sec
             0      cpu-migrations:u          #    0.000 K/sec
        75,254      page-faults:u             #    0.002 M/sec
94,585,862,412      cycles:u                  #    2.542 GHz
54,248,843,069      instructions:u            #    0.57  insn per cycle
<not supported>     branches:u
   647,868,750      branch-misses:u

  37.213653660 seconds time elapsed

  36.699832000 seconds user
   0.383356000 seconds sys
```

图 5.3: 并行

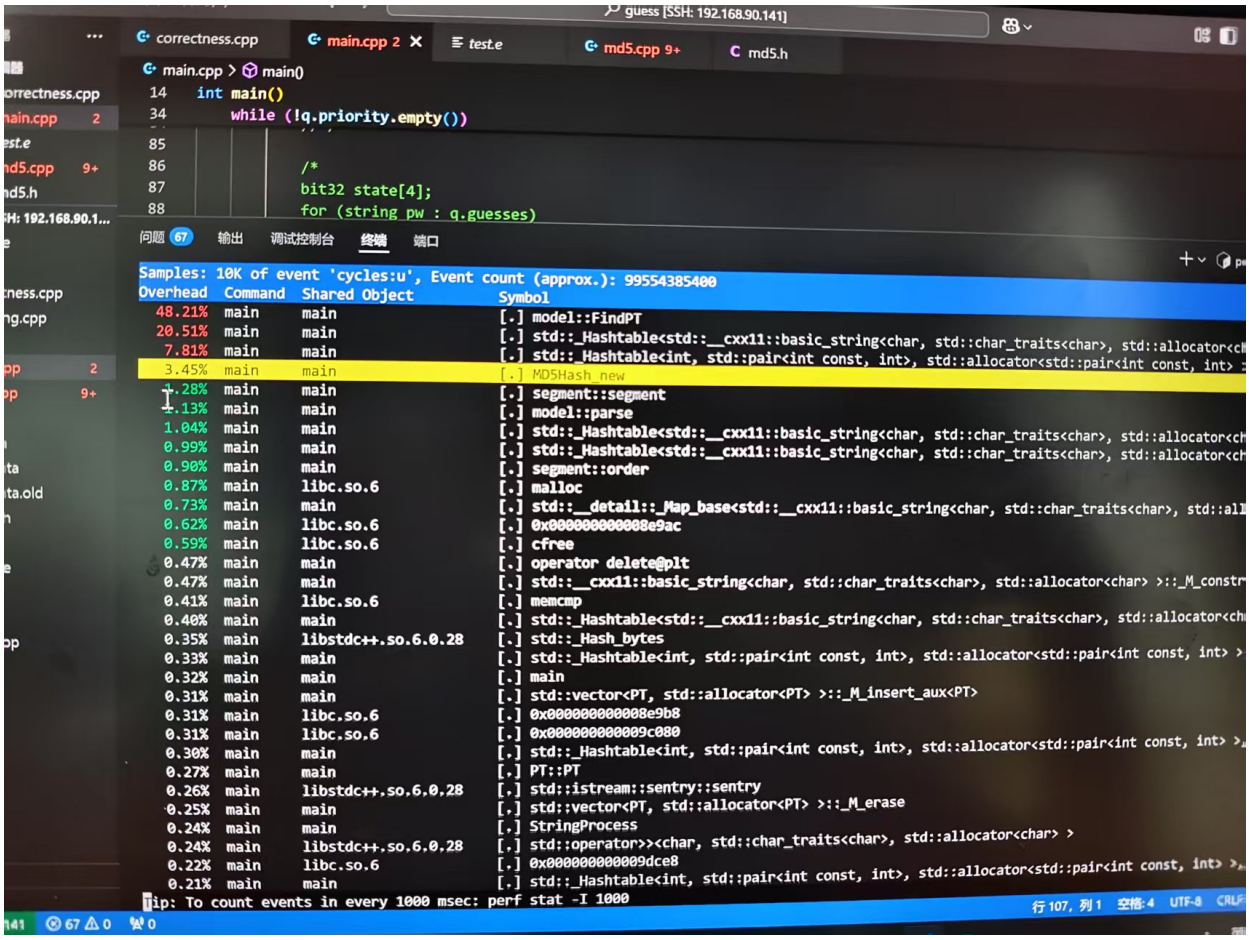


图 5.4: 并行

5.3 对比分析

在本次实验中，通过使用 perf stat 对比串行版与 SIMD 优化版 MD5 算法的性能数据，发现 SIMD 版本在 instructions、cpu 利用率以及程序占比等关键指标上均有明显改善。具体而言，指令数 instrutions 由 602 亿降低为 542 亿，cpu 利用率从 0.998 上升到 1，MD5 函数在整个口令猜测程序中占比由 4.08% 变为 3.45%，体现了 SIMD 优化后程序执行总时间缩短，指令总数显著下降，且每周期执行指令数（IPC）明显提高，表明处理器资源利用率得到提升。这些数据综合验证了 SIMD 并行计算通过一次处理多数据元素，降低指令数量、提高并行度，从而有效加速整体计算过程的原理。值得注意的是在整个程序的其它指标方面，SIMD 算法与并行算法相差不大，可能的原因是当前的并行实现未能有效利用硬件资源，反而因线程/向量单元间的协调开销、缓存一致性协议冲突或负载不均衡等问题，进一步的优化方向是重构并行任务划分和内存访问模式。

6 探究不同编译优化的影响

对于不同编译器的优化影响，我们将基于串行算法和处理数为 4 的并行算法来探究。如下表我们得到采取不同编译优化的实验数据。

分析实验数据我们可以发现，当不采取编译优化时，SIMD 并行并没有实现加速，采取编译优化后 SIMD 并行效率明显提升，成功实现了相对于串行的加速。通过计算可以得到，采用 O1 优化时的

	不编译优化	O1 优化	O2 优化
串行	9.837	3.159	3.185
SIMD 并行	14.535	2.086	2.044

表 2: 性能测试结果 (o1 优化)(单位:s)

加速比为 1.51，采用 O2 优化的加速比是 1.54。至此，我们可以初步等到一个结论，使用编译优化能够明显提高程序的执行效率，并且若实现了相对于串行的加速，加速比会有小幅提升。

针对这一现象，查阅资料，我们可以得知出现这一现象的可能原因。

在未开启编译优化（-O0）的情况下，并行 MD5 实现未能实现预期的加速效果，这主要是因为编译器在此模式下会保留完整的调试信息并禁用所有优化策略，导致生成的机器码效率低下。具体表现为：NEON 向量指令无法形成有效的指令流水线，频繁发生寄存器溢出到内存的情况，函数调用开销完全保留，内存访问模式保持源代码顺序，无法利用缓存预取等硬件特性。这些因素共同导致即使采用了并行计算策略，其性能提升也被底层实现的低效所抵消。

当采用-O1 基础优化级别时，编译器会启用一系列关键优化：首先进行基本的指令调度优化，使 NEON 指令能够形成较紧密的流水线；其次实施寄存器分配改进，显著减少 NEON 寄存器的内存溢出操作；同时对循环结构进行初步优化，并选择性内联小型 NEON 操作函数。这些优化使得计算密集型部分在整体执行时间中的占比显著提升，从而实现了可观测的加速效果。此外，-O1 还会简化调试信息并移除断言检查，进一步减少运行时开销。

而升级到-O2 优化级别后性能提升不明显，这可能是受制于以下几个因素：在算法特性上，MD5 的四轮处理过程存在严格的数据依赖链，每轮计算都依赖于前一轮的结果，严重限制了指令级并行的潜力；从硬件利用角度看，-O1 已经较好地开发了 NEON 指令集的并行计算能力，更激进的优化难以突破硬件并行单元的数量限制；在内存访问方面，MD5 的访存模式相对规整，-O1 已能较好地利用缓存层次结构；从控制流角度看，MD5 的主循环分支模式简单可预测，更高级别的分支预测优化收效甚微。实际上，在某些情况下-O2 可能会因为过度展开循环或激进内联而导致代码膨胀，反而可能降低指令缓存命中率。

7 总结

1. 针对 MD5 哈希算法的 SIMD 优化在使用编译优化的情况下能够实现加速，但总体来说，加速的效果不是特别明显。分析原因主要有（1）对于 MD5 哈希算法的理解不够深入，没有找打更好的并行优化方法，对于一些重要的步骤没有找到更好的算法；（2）可能是单次计算的并行度不够，或许可以提高到 8 个（时间紧迫，没有去具体验证）（3）初次接触并行编程，相关知识掌握不够。

2. 本次实验我了解了关于口令猜测算法的基本框架和实现，能够利用 neon 指令对于 MD5 哈希算法实现简单的并行化处理。除此之外，我学到 SIMD 的相关知识，知道 SIMD 即单指令流多数据流，是一种采用一个控制器来控制多个处理器，同时对一组数据中的每一个分别执行相同的操作从而实现空间上的并行性的技术，简单来说就是一个指令能够同时处理多个数据。