南開大學
Nankai University

计算机学院

并行程序设计实验报告

PCFG 口令猜测并行化选题: SIMD 实验

姓名：陈语童

学号：2311887

专业：计算机科学与技术

2025 年 4 月 29 日

目录

1 实验环境

1.1 SSH 远程：OpenEuler ARM 服务器

课程提供的 OpenEuler 服务器集群，以物理机 + 虚拟机的形式搭建的，目前提供给并行课程使用的共有 1 个主节点 (master_ubss1) 和 17 个计算节点 (master_ubss1 9, node1_ubss1 8)，每个计算节点 8 核。

OpenEuler 是开源、免费的 Linux 发行版操作系统，使用 ARM 架构指令集。

通过 vscode （或 cmd/powershell）进行相关 SSH 配置并进行连接，登入服务器中为学生个人分配的节点，进行 ARM 架构上的并行代码测试。

1.2 本地：Windows x86 PC

Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz.

L1 Cache = 64KB/Core, L2 Cache = 256KB/Core, L3 Cache = 8MB Shared.

2 问题描述

2.1 选题：PCFG 口令猜测

2.1.1 口令猜测

口令，就是俗称的“密码”。对于一定长度范围的口令，通过穷举进行暴力破解的效率是极低的。而本选题要做的，就是通过用户口令的语义和偏好规律，制定一定的策略来破解口令，并用**并行化**的方法对整个过程的各个部分进行优化与加速。

本选题中，不考虑个人信息、口令重用这些额外的信息，只考虑非定向的口令猜测。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。那么问题就变成了：

- 如何有效生成用户可能选择的口令；
- 如何将生成的口令按照降序进行排列。

在本选题中，选择使用最经典的 **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。

2.1.2 PCFG 串行训练

PCFG 串行训练流程关键步骤如下：

1. **内容分类**。将口令内容分为三类不同字段（segments）——Letters（字母字段）、Digits（数字字段）、Symbols（特殊字符）
2. **解构口令**。然后将实际每条口令内容用这三种字段类型解构其组成，形成一系列的口令 preterminal（类似于 pattern）。
3. **统计频率**。对不同的 preterminal，以及每个 preterminal 中的每个 segment 的不同值均需统计出现频次。

经过上面三步，PCFG 模型生成。

2.1.3 PCFG 串行猜测生成

猜测生成的本质要求，是构建出一个出现概率降序排列的口令**优先队列**，对其动态更新和从中取猜测值。具体生成策略较为繁琐，此处只做概括：

1. **初始化**。所有 preterminal 中出现概率最高的口令组合先入队列，preterminal 本身的概率越高越优先入队列。同时，赋给一个初始 pivot 值为 0.
2. **出列**。只要队列有口令，就从先入队列的开始取，取到的即作为当前口令猜测值。
3. **变异**。取出的口令不立即放回，而是根据 pivot 值，依次单项更新大于 pivot 值的 segment 值，依然是概率降序入队列，并且新入列的要设置 pivot 值到发生变异的 segment.
4. **遍历**。接着向后遍历和动态更新即可。

通过这种策略生成的优先队列可尽可能保证口令是不重复地以概率降序排列的，从而提高遍历查找效率和命中率。

2.1.4 PCFG 猜测生成并行化 *

这是本选题的重点和最终目标。

PCFG 猜测生成过程似乎比较复杂，难以进行并行化。并行化的最大阻碍，是按照概率降序生成口令这一过程。但 PCFG 往往用于没有猜测次数限制的场景中（例如哈希破解），实际应用中并不需要严格按照降序生成口令。

由此产生的并行化思路很显然：一次取出多个 preterminal，或是一次为某一 segment 分配多个 value 等。

选题代码框架采用的并行化方案如图示：

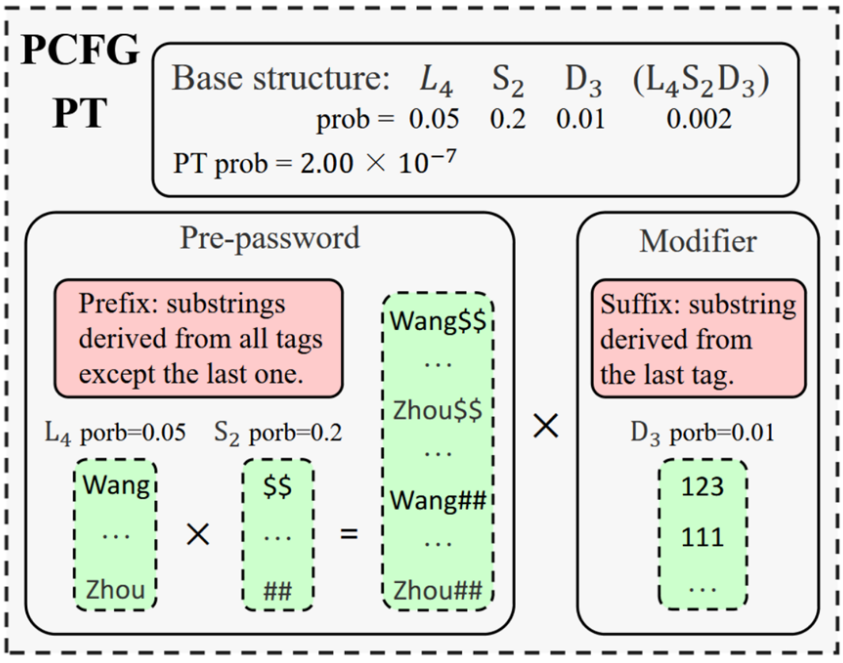

图 2.1: 本代码框架采用的并行化方案

这种方案和原始 PCFG 的区别在于，其对 preterminal 中最后一个 segment 不进行初始化和改变。在 preterminal 从优先队列中弹出的时候，直接一次性将所有可能的 value 赋予这个 preterminal. 不难看出，这个过程是可以并行进行的。

本选题的代码框架，已经将这个并行算法用串行的方式进行了实现，实际实验只需将其用并行编程的方式，变成一个并行程序，即可完成 PCFG 的并行化。

2.2 本次实验：SIMD 实验

PCFG 不太便于用 SIMD 进行并行化，选题额外添加了 MD5 来补足 SIMD 实验。

2.2.1 MD5 哈希算法的原理

（背景知识略）

MD5 信息摘要算法是一个常用的哈希算法/密码散列函数。对任意长度的信息（字符串，文件，图像等），MD5 都能为其生成一个 **128bit 即 16 字节的哈希值**。

简要介绍其生成过程：

1. 将信息转化为比特串，切割成若干 512bit 切片。
2. 从切片结果看，只要最后一个切片的长度不为 448bit，就需要进行补位。原则是，第一位为 1，后面全为 0，直到补位到产生了一个 448bit 的尾余切片，停止。
3. 最后 64bit 用二进制表示原始信息的长度，单位 bit. 从而，最终生成的比特串的长度应为 n*512bit. 至此，信息**预处理**完成。
4. 开始维护两个 128bit（每个分为四块，每块 32bit 即 4byte）的状态值缓冲区。对外循环缓冲区，首先初始化为预设值，分别为：a = 01 23 45 67, b = 89 ab cd ef, c = fe dc ba 98, d = 76 54 32 10.（这是大端序表示，实际维护过程中是小端序）
5. 接着在内循环缓冲区进行实际运算。引入之前的信息比特串 512bit 切片，将每个切片再细分为 16 个 32bit 的小块，用 32bit 小块的实际值，与一些常数变量一并传入预先定义好的一系列**位运算**函数，在外循环缓冲区 a，b，c，d 值的基础上作特定处理。每一小块都进行一次函数调用的处理，即一轮运算有 16 次处理，**前后顺序有依赖**。
6. 每一轮运算完成后，内循环缓冲区得到的 a，b，c，d 结果**加回到**外循环缓冲区内对应值上，外循环缓冲区就完成这一轮的更新。n 个 512bit 切片全部运算完毕（最后一轮也要加回），外循环缓冲区内的 a，b，c，d 进行顺序拼接，形成的 **128bit 即 16 字节的哈希值**就是该信息进行 HD5 的最终结果。

2.2.2 MD5 哈希算法的并行化

对于单个信息的 MD5 计算过程，难以进行并行化。所以本实验的并行化思路主要在于对多个信息同时进行 MD5 计算。

事实上，MD5 的运算在数据上是**高度对齐**的，并且运算过程具有很高的确定性。再观察到一系列缓冲区处理函数的定义（此略），发现其运算不涉及条件判断，且为较简单的运算（移位、与运算，etc.）。这就使得这些函数非常适合用 SIMD 进行并行化处理：一次性地让这些**函数同时处理多个口令**，即可实现并行化。

3 基础实验

基础要求：试在 ARM 服务器上，基于 NEON 实现 MD5 哈希算法。具体要求如下：

- 要求有基本的正确性，即能够利用 SIMD 指令，做到一次性产生多个消息（口令）的哈希值，并且哈希值正确；
- 不要求真正实现相对串行算法的加速，但是仍需要给出实验数据，并且分析为什么没有能够实现加速。

3.1 基于 ARM NEON 的 MD5 并行化实现

3.1.1 并行度分析与选择

通过上面的分析，已经确定了 SIMD 并行化的方向。现在进入到代码编写阶段，首先应根据编程环境确定合适的并行度。

NEON 指令集提供了一系列向量数据类型，可根据总位宽、向量单个元素位宽、向量维数等进行划分，此处不一一列举。按照总位宽可分为 64bit 和 128bit 两种，为了尽可能提高并行度，自然选择 128bit 类型的向量。而根据 MD5 的生成原理，每个信息最终的哈希值最小的划分是 4 个 32bit 的无符号整型数，简单计算就可以得出，128bit 位宽的向量恰好容纳 4 个元素。

因此可以确定并行度：$PARA\_NUM = 4$, 对应的向量类型：`uint32x4_t`.

3.1.2 并行化代码实现

MD5 并行化代码实现的关键在于 [md5.cpp] 中 `MD5Hash` 函数的 SIMD 并行迁移，以及相关辅助函数的修改。

首先看 MD5 的主要逻辑部分，最基本的操作是并行化哈希值的生成过程，具体来说就是将相应的变量转换成 NEON 提供的数据类型，并使用 NEON 指令进行各类赋值操作。以下是对于原来 [md5.cpp] 中 MD5 实现过程 `MD5Hash` 函数的 SIMD 四路并行优化 `SIMDMD5Hash_4` 函数的伪代码：

Algorithm SIMDMD5Hash_4

function SIMDMD5Hash_4(inputs, state)
    PARA_NUM ← 4 ▷ 设置并行度
    messageLengths ← new array of size **PARA_NUM** ▷ 存储处理后消息长度（单位：bit）
    maxByte ← 0 ▷ 初始化处理后比特串最大值，实际上在本实验中无需检测最大值
    paddedData ← SIMDStringProcess(**inputs**, **messageLengths**, **PARA_NUM**, **maxByte**)
        ▷ 生成一个经过**地址对齐**处理的整合数据块，顺序存储 4 个口令的比特串
    n_blocks ← **messageLengths[0]** / 64 ▷ 将比特串划分为 n_block 个 512bit 大小的切片

    /* 初始化 `uint32x4_t` 向量类型的哈希值缓冲池 */
    state_a ← $[0x67452301, 0x67452301, 0x67452301, 0x67452301]$
    state_b ← $[0xefcdab89, 0xefcdab89, 0xefcdab89, 0xefcdab89]$
    state_c ← $[0x98badcfe, 0x98badcfe, 0x98badcfe, 0x98badcfe]$
    state_d ← $[0x10325476, 0x10325476, 0x10325476, 0x10325476]$

    /* 循环：`MD5` 核心逻辑 */
    for $i = 0$ **to n_blocks - 1 do**

```
        for i1 = 0 to 15 do
            Load M[i1] with 4 parallel 32-bit words from paddedData   ▷ M[16] 是 uint32x4_t 类型向量，存储 16 个 32bit 大小的 512bit 切片的子切片
        end for
      // 初始化内循环哈希值缓冲池
        for i = a, b, c, d do
            i ← state_i
        end for
      // 调用 SIMD 并行化版本的位操作函数，此为 MD5 核心处理部分
        Perform Round 1 with FF_SIMD
        Perform Round 2 with GG_SIMD
        Perform Round 3 with HH_SIMD
        Perform Round 4 with II_SIMD
      // 每轮处理完成后内循环得到的哈希值加回到外部缓冲池进行更新
        for i = a, b, c, d do
            state_i ← state_i + i
        end for
    end for

    /* 转置，便于结果的输出 */
    for i = 0 to 4 do
        state_i ← [state_a[i], state_b[i], state_c[i], state_d[i]]
    end for

    /* 传给结果哈希值 */
    Store state_0, state_1, state_2, and state_3 into state

    /* 大端格式化 */
    for i = 0 to 4 * PARA_NUM - 1 do
        Convert state[i] to big-endian format
    end for

    /* 回收内存 */
    Free paddedData
    Free messageLengths
end function
```

对于以上代码逻辑，需作几点说明：

- **uint32x4_t：** `uint32x4_t` 是 ARM NEON 指令集里的**向量数据类型**，表示“一个包含 4 个无符号 32 位整数的 SIMD 向量”。缓冲池和子切片，以及读取数据等操作的中间变量均声明为这种向量数据类型，保证所有处理都同时改变向量中的 4 个元素，从而达到同时处理 4 组信息的 **SIMD 并行化**目的。
- **paddedData：** 函数设计中，选择将 4 条信息整合为一个地址对齐的完整内存块，前提是各条消息长度相等，而恰好有此前提。具体实现，通过一个新增的 `SIMDStringProcess` 函数来实现，是

对于原先代码已有函数 `StringProcess` 的改写。以下是 `SIMDStringProcess` 函数的伪代码：

Algorithm SIMDStringProcess

```
1: function SIMDStringProcess(inputs, n_byte, guess_num, max_byte)
2:     ALIGNMENT ← 16                                  ▷ ARM NEON 要求 16 字节的地址对齐
3:     maxPaddedLength ← 0          ▷ 所有信息中处理后最大的字符串长度（单位：字节）
4:     paddedLengths ← new array of size guess_num

       /* 获取比特串最大长度 */
5:     for i ← 0 to guess_num − 1 do
6:         paddedLengths[i] ← CalculatePadded(length of inputs[i]) ▷ 求长度再作单独封装
7:         if paddedLengths[i] > maxPaddedLength then
8:             maxPaddedLength ← paddedLengths[i]
9:         end if
10:    end for

       /* 分配对齐的内存块 */
11:    paddedData ← aligned_alloc(ALIGNMENT, maxPaddedLength × guess_num)

       /* 核心逻辑：比特串补位到 512bit 整数倍长度 */
12:    for i ← 0 to guess_num − 1 do
13:        blocks ← byte array of inputs[i]
14:        length ← length of inputs[i]
15:        paddedLength ← paddedLengths[i]
16:        paddedMessage ← paddedData + i× maxPaddedLength
17:        Copy(blocks[0..length-1] into paddedMessage[0..length-1])   ▷ 迁移原始数据
18:        paddedMessage[length] ← 0x80                                ▷ 补位的第一位是 1
19:        Fill(paddedMessage[length+1..paddedLength-9] with zeros)    ▷ 之后都是 0
       // 最后 64bit 记录原始信息长度（单位：bit）
20:        for i1 ← 0 to 7 do
21:            paddedMessage[paddedLength−8+i1] ← ((length × 8) ≫ (i1 × 8)) ∧ 0xFF
22:        end for
       // 检查内存块地址对齐
23:        Assert((8 × paddedLength) mod 512 = 0)
24:    end for

       /* 比特串长度和最大长度传给外部变量 */
25:    n_byte ← paddedLengths
26:    max_byte ← maxPaddedLength

       /* 回收内存 */
27:    Free paddedLengths
28:    return paddedData
29: end function
```

在函数当中，调用的 `CalculatePadded` 函数，是为了使代码整体逻辑层次更清晰而专门再封装的用来求**处理后比特串长度**的函数方法。此函数的逻辑与原先 `StringProcess` 中相应部分的逻辑完全一致，此处不再展示伪代码。此外，关于为何要用整合的内存块，而不是选择像 `paddedMessage[4]` 这样声明 `Byte` 数组，一部分原因是原本程序设计上的疏漏，没有注意到实验情境下的口令都是等长的，认为必须要通过整合块来对齐；另一部分原因是，后来发现用整合内存块进行处理比用 `Byte` 数组有约 0.4s 的运行时间缩减，分析其原因是直接声明 `Byte` 数组的地址不一定连续，从而在并行处理时访存更加困难。综合以上两个原因，保留了 `paddedData` 整合块的设计。

- **maxByte：** `SIMDMD5Hash_4` 和 `SIMDStringProcess` 中都有声明用来**记录最大比特串长度**的变量，且有提到实际操作中并不需要此变量。原本，这是一种为了保证整合内存块对齐的安全策略：假设实验中的口令长度并不完全一致，可能会产生不同长度的比特串，这时如果还想要保证能够四路并行处理，就需要取**最大比特串长度**来构建整合内存块，保证地址的对齐。这本来是一个程序设计上的疏漏（与上条说明同理），但后来考虑到一个多余的 int 型变量对整体的运行表现影响微乎其微，且如果将这个变量编写进后续的部分代码展示出的逻辑更清晰，因而保留了此变量。

MD5 主逻辑的 SIMD 实现完整代码详见 GitHub 仓库：[md5.cpp]

此外，对于 MD5 的核心处理逻辑中的 `FF/GG/HH/II` 函数也要进行相应更改，迁移到 NEON 指令集。以下是以 `FF` 函数为例的修改，在 [md5.h] 中：

```
#define F_SIMD(x, y, z) vorrq_u32(vandq_u32(x, y), vandq_u32(vmvnq_u32(x), z))

#define ROTATELEFT_SIMD(num, n) \
    vorrq_u32(vshlq_n_u32((num), (n)), vshrq_n_u32((num), (32 - (n))))

#define FF_SIMD(a, b, c, d, x, s, ac) { \
  a = vaddq_u32(ROTATELEFT_SIMD(vaddq_u32(vaddq_u32(a, F_SIMD(b, c, d)), \
      vaddq_u32(x, vdupq_n_u32(ac))), s), b); \
}
```

实际上，唯一的修改即将基础的四则运算和位运算全部替换为适配于 `uint32x4_t` 向量的相应 NEON 指令，如 `vaddq_u32`（加法），`vorrq_u32`（或），`vshlq_n_u32`（左移），`vshrq_n_u32`（右移）等。

所有核心处理函数的 SIMD 实现完整代码详见 GitHub 仓库：[md5.h]

3.2 正确性检验

给 [correctness.cpp] 新增并行化的测试，调用 `SIMDMD5Hash_4`，进行哈希值正确性的检测。

测试结果表明正确，截图见 GitHub 仓库：[correctness -0 -2 -4 -8(-) -8(+).jpg]

3.3 SIMD 优化的加速效果

[main.cpp] 已经使用 `system_clock` 标记了 MD5 Hash 过程的开始和结束时刻，从而计算整个哈希过程的用时。在主函数中添加 `SIMDMD5Hash_4` 函数的调用逻辑，先后选择原**串行**算法和改进后的 **SIMD 四路并行**算法，进行编译、运行测试，收集并观察输出结果中的 `Hash time` 参数。

多次测试，去除最高值最低值的结果如下：

no.	1	2	3	4	5	**avg.**
serial	2.83988	2.97881	2.93737	3.08496	2.9881	**2.9658**
SIMD*4	1.68335	1.67164	1.66344	1.68123	1.66313	**1.6726**

表 1: 串行和四路并行 MD5Hash 用时（单位：秒）

根据均值结果可以求得四路并行算法相较于串行算法在 MD5Hash 上的**加速比**：

$$\text{Speedup(SIMD*4)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{SIMD*4})} = \frac{2.9658}{1.6726} \approx 1.77 > 1$$

从结果来看，SIMD 四路并行改进对于 MD5Hash 过程有较为显著的**加速效果**。

4 进阶实验

4.1 实现相对串行算法的加速

基础实验中已经实现，见 3.3 节.

4.2 探究编译选项对于加速比的影响

4.2.1 优化选项（无/-O1/-O2）

前述实验都是以**-O2 最高优化**下编译运行的，接下来探究不同优化选项下对 SIMD 四路并行算法相对于串行算法的加速比有什么影响。

增加两大组实验，分别选用无优化和-O1 优化选项，对 SIMD 算法和串行算法分别进行多组实验去除最低最高值，得到的结果如下表所示：

*type**no.*		1	2	3	4	5	**avg.**
$-O0$	serial	9.66974	9.4504	9.39494	9.75502	9.75654	**9.6053**
	SIMD*4	15.0711	15.0542	14.9883	15.1395	15.0474	**15.0601**
$-O1$	serial	3.22641	3.00129	3.11881	3.02099	3.0412	**3.0817**
	SIMD*4	1.84643	1.84516	1.84338	1.8416	1.84022	**1.8434**
$-O2$	serial	2.83988	2.97881	2.93737	3.08496	2.9881	**2.9658**
	SIMD*4	1.68335	1.67164	1.66344	1.68123	1.66313	**1.6726**

表 2: 不同优化选项下串行和四路并行 MD5Hash 用时（单位：秒）

根据均值分别计算三种优化选项下的加速比：

$$[-O0]: \quad \text{Speedup(SIMD*4)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{SIMD*4})} = \frac{9.6053}{15.0601} \approx 0.64$$

$$[-O1]: \quad \text{Speedup(SIMD*4)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{SIMD*4})} = \frac{3.0817}{1.8434} \approx 1.67$$

$$[-O2]: \qquad \text{Speedup(SIMD*4)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{SIMD*4})} = \frac{2.9658}{1.6726} \approx 1.77$$

很显然，加速效果有：$\underline{[-O2] > [-O1] > [-O0]}$. ($-O0$ 就是无优化选项) 注意到无优化选项下的 SIMD 四路并行算法加速比 < 1，为负加速比，即非但没有实现加速，反而还使哈希过程变慢了。接下来对于上面的结果逐一分析：

Q：为什么 -O0 时呈负加速比？

A：无优化选项的条件下，很多 SIMD 的 NEON 指令都没有真正实现功能，具体来说可能有以下几个方面的问题：

- 内存对齐和访存策略不佳：未经优化的编译可能无法保证数据（如 `paddedData`）在内存中的对齐，或者不根据 SIMD 的地址对齐规则进行访存，导致 SIMD 操作处理数据时更频繁地发生内存访问冲突（cache misses 等），从而降低性能。
- 指令流水线阻塞：编译器未重排指令，SIMD 指令与串行代码之间的数据依赖性导致流水线频繁停顿。例如，NEON 寄存器操作与普通寄存器间的数据搬运未优化，产生冗余指令等。
- 不充分的寄存器利用：无优化时，编译器可能没有充分利用寄存器来让 SIMD 指令并行化地处理数据。可能存在不必要的寄存器分配，导致更频繁的数据存取或寄存器溢出。
- 冗余计算保留：一些重复类计算未被优化，导致冗余计算。例如，循环内的临时数组 `temp_vec` 在每次迭代中重新分配，paddedData 的索引重复计算等。
- 函数调用开销：函数未被内联，跳转指令和参数传递提高延迟。
- 指令调度策略不佳：无优化时，编译器可能没有重新安排相关指令的执行顺序来避免流水线冲突或空转周期，也就是降低了 SIMD 指令的实际性能。

Q：-O1 和 -O2 的优化分别体现在哪？

A：-O1 和-O2 的根本目标都是提高运行效率，减少运行时间。简单概括而言，-O2 是相较于-O1 **更激进和完全**的优化。具体来看，-O1 的优化包括：（短）循环展开[1]，循环不变代码外提[2]，（短）函数内联，常量传播[3]，基本内存对齐优化 $\cdots\cdots$ -O2 的优化包括：完全的循环展开，循环分块[4]，自动向量化[5]，跨函数优化[6]，更完全的内联，寄存器分配优化，指令调度[7] $\cdots\cdots$

Q：综合分析三阶段的加速提升？

A：首先从 -O0 到 -O1，实现了从无加速效果到有显著加速效果的飞跃，主要是来自于**循环展开**（减少分支开销）和**内存对齐**（减少缓存未命中）的 -O1 优化，以及基本的**内存对齐**，这使得 SIMD 指令发挥了应有的作用；其次从 -O1 到 -O2，加速比又有小幅度提升，主要是来自于**指令调度**优化，以及各方面相较于 -O1 更加**激进和完全**的 -O2 优化（**循环**、**函数**等方面），此外**自动向量化**可能也发挥了作用。

[1] **循环展开：** 将循环展开成循环轮数的同一段代码，减少循环控制开销。

[2] **循环不变代码外提：** 将循环内不依赖迭代变量的计算移到循环外，减少重复计算。

[3] **常量传播：** 将编译期已知的常量直接替换到代码中，减少运行时计算。

[4] **循环分块：** 将大循环拆分为小块，提高缓存局部性。

[5] **自动向量化：** 尝试将标量代码自动转换为 SIMD 指令（如 NEON），即使没有手动编写 SIMD 代码。本实验中理论上已经手动实现了足够的并行向量化，优化本身并不依赖自动向量化。

[6] **跨函数优化：** 分析函数之间的交互（如逻辑方法函数和主函数之间），优化内存分配和数据传递。

[7] **指令调度：** 重新排列指令顺序，实现重叠执行等，隐藏 SIMD 指令的延迟。

4.2.2 其他编译指令

查阅关于 G++ 编译选项，有 `-fopenmp`，`-funroll-loops`，`-ffast-math` 等其他编译指令，但尝试后均不能再提升加速比，此处不再作数据分析。

4.3 x86 SSE 本地迁移

现尝试将在 OpenEuler 上完成的 SIMD 并行化 MD5 哈希过程迁移到 x86 本地，换用 SSE 指令完成 SIMD 四路并行.

一些需要使用到的 NEON 和 SSE 指令转化对照如下表所示：

usage	NEON	SSE
128bit 四维向量类型	`uint32x4_t`	`_m128i`
向量元素赋值	`vdupq_n_u32(c)`	`_mm_set1_epi32(c)`
从内存加载数据	`vld1q_u32(ptr)`	`_mm_loadu_si128((__m128i*)(ptr))`
将数据存储到内存	`vst1q_u32(ptr, a)`	`_mm_storeu_si128((__m128i*)(ptr), a)`
提取第 i 个元素	`vgetq_lane_u32(a, i)`	`_mm_extract_epi32(a, i)`
逐元素无符号整数加法	`vaddq_u32(a, b)`	`_mm_add_epi32(a, b)`
逐位与操作	`vandq_u32(a, b)`	`_mm_or_si128(a, b)`
逐位或操作	`vorrq_u32(a, b)`	`_mm_add_epi32(a, b)`
逐位取反	`vmvnq_u32(a)`	`_mm_andnot_si128(a, _mm_set1_epi32(-1))`
位运算左移	`vshlq_n_u32(a, n)`	`_mm_srli_epi32(a, n)`
位运算右移	`vshrq_n_u32(a, n)`	`_mm_slli_epi32(a, n)`

表 3: NEON 指令和 SSE 指令对照表

对照上面的表格修改 [md5.cpp] 和 [md5.h] 中的 SIMD 指令，移动到本地后进行编译和运行，发现运行到 `SIMDMD5Hash` 函数内部无法正确返回。经过调试，发现问题在于迁移 NEON 到 SSE 之后，原本对于 `paddedData` 的管理体系不能适配，程序无法解决相关内存分配故障。因此退而求其次地使用 `Byte` 类型数组的口令比特串管理模式：

Algorithm SIMDMD5Hash

```
1: function SIMDMD5Hash(inputs, state)
2:     PARA_NUM ← 4                                   ▷ 并行度为 4
3:     messageLengths ← new int array of size PARA_NUM
4:     paddedMessages ← array of 4 pointers to Byte arrays   ▷ 换用 Byte 数组管理口令比特串
       /* 后续主体逻辑基本一致，此处略 */
```

策略更改后的 SSE 版本 `SIMDMD5Hash` 函数直接调用原本的 `StringProcess` 函数，无需编写新的辅助函数。

SSE 版本的 SIMD 四路并行 MD5 方法函数 `SIMDMD5Hash` 函数完整代码见 GitHub 仓库：[md5.cpp]

相应地要修改一系列位操作函数，完整代码见 GitHub 仓库：[md5.h]

完整代码通过 [correctness.cpp] 进行正确性检验，检验结果正确，截图见 GitHub 仓库：[SIMD4.jpg]

编译运行后发现，本地运行 SSE 迁移后的 MD5 所需时间平均仅需约 1.4341 秒（1.45325s 1.43147s 1.43517s 1.42402s 1.43068s 五次有效数据求均值得），比 OpenEuler 上测试的时间（平均约 1.6726s）还要短。猜测是服务器有任务竞争资源，或是有延迟等原因，有待验证。

4.4 perf 性能分析

perf 是 Linux 下一个强大的性能分析工具，利用其可对程序进行综合性能分析、软硬件事件跟踪等任务从而对所编写的程序有更全面深入的了解。

本次实验中，使用如下命令，在运行编译完成的 `main` 程序的同时进行 perf 性能分析：

```
perf stat -e
   cache-misses,cache-references,L1-dcache-load-misses,L1-dcache-loads,LLC-load-misses,LLC-loads
      bash test.sh 1 1
```

其中 `cache-misses` 获取缓存引用未命中数，`cache-references` 获取缓存总访问数，`L1-dcache-load-misses` 获取一级/L1 数据缓存未命中数，`L1-dcache-loads` 获取一级/L1 数据缓存总访问数，`LLC-load-misses` 获取末级（一般是三级/L3）数据缓存未命中数，`LLC-loads` 获取末级数据缓存总访问数。

从若干分析结果来看，总访问数**总是**等于一级访问数，这似乎不合逻辑，因为总访问数应该统计所有级别缓存的访问数，猜测可能是 perf 工具设置有误。除此之外，无论是串行算法还是 SIMD 并行化算法，L1 未命中率都在 1% 上下浮动，SIMD 并行化算法一般略小于串行算法但基本持平；两者 L3 未命中率浮动幅度较大，一般在 20% 和 30% 之间浮动，但几乎总是有 SIMD 算法的未命中率小于串行算法的未命中率。然而，一般认为 L3 未命中率超过 20%，该指标并不属于优秀范围，因此 SIMD 并行化设计并没有给**数据访问局部性**以及**数据利用率**带来很大的提升，应在后续实验中注意这方面的优化。

完整的 perf 分析结果见 GitHub 仓库：[perf res.md]

4.5 改变并行度

SIMD 可以控制“单指令多数据”中，数据的总数。例如，可以一次计算两个、四个、八个消息的哈希值。此小节尝试改变单次运算的并行度，探索其加速效果会发生怎样的改变。

4.5.1 二路并行

四路并行使用了 128bit 大小的向量，而二路并行只需要一半，即 64bit. 查阅后发现 NEON 提供此大小的向量类型，其中有 `uint32x2_t`，即两个 32bit 元素的向量，恰好满足需求。

原先的位运算函数是针对 `uint32x4_t` 向量类型处理的，不再适用，需要在 [md5.h] 中重新编写针对 `uint32x2_t` 的版本。下面是一个示例：

```
#define F_SIMD_2(x, y, z) vorr_u32(vand_u32(x, y), vand_u32(vmvn_u32(x), z))

#define ROTATELEFT_SIMD_2(num, n) \
    vorr_u32(vshl_n_u32((num), (n)), vshr_n_u32((num), (32 - (n))))

#define FF_SIMD_2(a, b, c, d, x, s, ac) { \
```

```
  a = vadd_u32(ROTATELEFT_SIMD_2(vadd_u32(vadd_u32(a, F_SIMD_2(b, c, d)), vadd_u32(x,
  ↪ vdup_n_u32(ac))), s), b); \
}
```

其他修改较为简单，不再展示伪代码，完整代码实现见 GitHub 仓库：[md5.cpp]

4.5.2 八路并行

由于 NEON 指令架构下，向量最大只有 128bit，而八路并行需要同时处理 256bit 大小的口令比特串数据，必须制定一个能够满足这一要求的方案。经过探索，确定了以下两种可行方案：

(1) 高低位向量→ SIMDMD5Hash_8basic

一个比较朴素的想法是，将八条信息分为“高/低”两组，每组四条消息，然后为高低组分别分配向量，将四路并行的所有操作翻倍，就可以实现八路并行。体现在代码上，就是所有声明和操作有 `_low` 和 `_high` 两部分。由于此方案使用的仍然都是 `uint32x4_t` 向量，原来的位运算函数仍然适配，无需修改。报告篇幅限制，此处不展示伪代码，完整代码实现见 GitHub 仓库：[md5.cpp]

(2) 复合向量→ SIMDMD5Hash_8advanced

询问 AI Agent 获得一个更高级的方案：使用复合向量声明。`uint32x4x2_t` 代表两个 128bit(32bit*4 元素）向量组合而成的复合向量，可以通过 `.val[0]`, `.val[1]` 的方式来取出其中的前一和后一向量。相对于前一种朴素的方案，这一方案使用的数据存储方式集成性更高，**声明**时和**调用位运算函数**时也不再需要翻倍，对比之下节省了不少的代码空间和额外时间开销。

同样考虑位运算函数，原本的位运算函数是针对 `uint32x4_t` 向量类型处理的，此方案下不再适用，需要在 [md5.h] 中重新编写针对 `uint32x4x2_t` 的版本。下面是一个示例：

```
#define F_SIMD_8advanced(x, y, z) (uint32x4x2_t){ \
  vorrq_u32(vandq_u32((x).val[0], (y).val[0]), vandq_u32(vmvnq_u32((x).val[0]), (z).val[0])), \
  vorrq_u32(vandq_u32((x).val[1], (y).val[1]), vandq_u32(vmvnq_u32((x).val[1]), (z).val[1])) \
}

#define ROTATELEFT_SIMD_8advanced(num, n) (uint32x4x2_t){ \
  vorrq_u32(vshlq_n_u32((num).val[0], (n)), vshrq_n_u32((num).val[0], (32 - (n)))), \
  vorrq_u32(vshlq_n_u32((num).val[1], (n)), vshrq_n_u32((num).val[1], (32 - (n)))) \
}

#define FF_SIMD_8advanced(a, b, c, d, x, s, ac) { \
  uint32x4x2_t tmp; \
  tmp.val[0] = vaddq_u32((a).val[0], vaddq_u32(F_SIMD_8advanced(b, c, d).val[0],
  ↪ vaddq_u32((x).val[0], vdupq_n_u32(ac)))); \
  tmp.val[1] = vaddq_u32((a).val[1], vaddq_u32(F_SIMD_8advanced(b, c, d).val[1],
  ↪ vaddq_u32((x).val[1], vdupq_n_u32(ac)))); \
  tmp = ROTATELEFT_SIMD_8advanced(tmp, s); \
  tmp.val[0] = vaddq_u32(tmp.val[0], (b).val[0]); \
  tmp.val[1] = vaddq_u32(tmp.val[1], (b).val[1]); \
  (a).val[0] = tmp.val[0]; \
  (a).val[1] = tmp.val[1]; \
}
```

其他修改较为简单，不再展示伪代码，完整代码实现见 GitHub 仓库：[md5.cpp]

二路和八路并行也进行了 [correctness.cpp] 的验证，结果正确，截图见 GitHub 仓库：[correctness -0 -2 -4 -8(-) -8(+).jpg]

4.5.3 加速效果综合对比

控制变量，让各并行度的算法都在 -O2 的编译条件下编译、运行，获得多次测试结果，去除最高最低值，获得以下有效测试值：

no.	1	2	3	4	5	**avg.**
serial	2.83988	2.97881	2.93737	3.08496	2.9881	**2.9658**
SIMD*2	3.14893	3.15891	3.13497	3.1324	3.13298	**3.1416**
SIMD*4	1.68335	1.67164	1.66344	1.68123	1.66313	**1.6726**
SIMD*8(-)	1.20376	1.19794	1.20582	1.2002	1.19746	**1.2010**
SIMD*8(+)	1.18629	1.18236	1.18812	1.1833	1.18762	**1.1855**

表 4: 各并行度下 MD5Hash 用时（单位：秒）

其中 SIMD*8(-) 是**高低位**向量的八路并行（`SIMDMD5Hash_8basic`），SIMD*8(+) 是**复合**向量的八路并行（`SIMDMD5Hash_8advanced`）。

分别计算各并行度下的加速比：

$$\text{Speedup(SIMD*2)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{SIMD*2})} = \frac{2.9658}{3.1416} \approx 0.94$$

$$\text{Speedup(SIMD*4)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{SIMD*4})} = \frac{2.9658}{1.6726} \approx 1.77$$

$$\text{Speedup(SIMD*8 [-])} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{SIMD*8 [-]})} = \frac{2.9658}{1.2010} \approx 2.47$$

$$\text{Speedup(SIMD*8 [+])} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{SIMD*8 [+]})} = \frac{2.9658}{1.1855} \approx 2.50$$

很显然，加速效果有：八路并行 > 四路并行 > 二路并行。其中注意到几点：

- 二路并行没有加速效果，甚至还略慢于串行算法。分析可能原因有二，一是 NEON 的数据装载过程的额外时间开销较大，且就相比于四路而言多了一倍的装载次数；二是 `uint32x2_t` 的处理效率可能是较低的。上面两个原因带来的额外时间消耗综合起来抵消了 SIMD 的并行优化效果。
- 八路并行的加速效果显著高于四路并行，近乎四路并行加速效果的 1.5 倍，符合期望。没有达到纯粹理想状态的提升一整倍，是因为处理更大体量的向量消耗的内存空间和时间也是要翻倍甚至更多的。
- 复合向量的方案比高低位向量的方案还要快一点，这在上面分析方案时已经提及原因，与集成性更高的复合向量可以节省代码空间和额外的访问时间开销等有关。

可见，四路和八路并行的优化都是十分成功的。

5 总结思考

这是本学期并行程序课程大作业的第一次实验，主要完成了对于 MD5 哈希算法的 SIMD 并行化优化，并且探索了一系列诸如编译、并行度、指令集等方面差异对于优化最终的效果，即相对于平凡的串行算法的加速比。

整体来说实验是成功的，对于各种可能性都作了较为完整和深入的尝试和探讨。在实验过程中也锻炼了 C++ 代码的编写能力，以及对于 SIMD 指令的运用技巧。

本次实验所有代码均已上传至 GitHub：

{OpenEuler 服务器中的 NEON 实现}

{x86 本地的 SSE 实现（仅四路并行）}

为证明数据真实性，在这里提供每组 SIMD 算法运行测试中时间最小的结果截图：[min data.md]