计算机学院

并行程序设计第四次报告

NTT 为例的 Pthread+OpenMP 编程

姓名：宋卓伦

学号：2311095

专业：计算机科学与技术

2025 年 5 月 29 日，南开大学计算机学院，天津

目录

1 实验目的及实验介绍

1.1 实验目的

经过这段时间对 Pthread 和 OpenMP 的学习，我对在这两种指令集下面进行编程有了一定的体会。前面的 SIMD 架构下的 NEON 指令集能够在一定程度上实现程序的加速和硬件性能的提升，在上课的时候我不禁产生了思考：如果把老师课堂上讲到的内容利用在我们的实验中？接下来，我将通过后面的 Pthread 编程和 OpenMP 编程来熟悉这两套编程的架构，解答前面的问题。

1.2 Pthread

Pthread 作为 POSIX 线程标准，**为开发者提供了规范且强大的线程编程接口**。其应用场景十分广泛，不仅被 Unix、Linux、macOS 等类 Unix 系统所采用，在 Windows 系统中也有相应的移植版本。

Pthread 的 API 命名规则与常规 C/C++ 代码保持一致，这一设计使得编程过程更易理解和上手，对成与原友好。以线程创建为例，使用的 pthread_create 函数包含多个参数，例如指向线程标识符的指针、线程属性、线程执行函数的起始地址以及运行函数的参数。通过这些参数，能够实现对线程创建过程的灵活控制，后续也会频繁利用这一特性。

在类 Unix 系统中，**Pthread 是多线程编程的基石**。以 Linux 系统为例，它对 Pthreads 提供了广泛支持，开发者可借助 Pthread 库在 Linux 环境中完成多线程的创建、管理与同步操作。而 macOS 作为苹果公司的操作系统，同样遵循 POSIX 标准，因此也对 Pthreads 提供了支持。

1.3 OpenMP

OpenMP（Open Multi-Processing）作为并行计算领域的关键标准，为开发者提供了简洁高效的多线程编程模型。它通过编译器指令与库函数结合的方式，支持 C、C++ 和 Fortran 等主流编程语言，广泛应用于高性能计算、数据科学、工程模拟等场景。无论是 Linux、Windows 还是 macOS 系统，OpenMP 都能提供统一的编程接口，**大幅降低了跨平台并行开发的难度**。

OpenMP 的设计理念聚焦于 **"以最小代码改动实现并行加速"**，核心在于通过 #pragma omp 等编译制导指令在源代码中嵌入并行区域定义。例如，利用 #pragma omp parallel for 指令可轻松将串行循环转换为并行执行，编译器会自动处理线程创建、负载均衡及结果合并等复杂操作。此外，OpenMP 还具备任务并行、同步机制、数据环境管理等高级特性，能够满足不同应用场景的需求。

在高性能计算领域，OpenMP 已成为共享内存系统并行编程的事实标准。**以 Linux 平台为例，GCC、Clang 等主流编译器均对 OpenMP 提供完善支持，开发者通过简单的编译选项（如fopenmp）即可启用并行功能**。对于 Windows 用户，Visual Studio 等开发工具同样集成了 OpenMP 支持，进一步提升了跨平台代码移植的便捷性。这种跨平台的一致性与易用性，使 OpenMP 成为学术研究和工业开发中并行编程模型的首选之一。

1.4 实验硬件和环境

本次实验选用远程 WSL 连接 Ubuntu24.04 进行本地编程（表1, 同时采用助教学长们搭建的 OpenEuler 服务器进行代码设计和主要的性能查看。

属性	相关内容
内核版本	5.15.167.4-microsoft-standard-WSL2
硬件架构	x86_64（64 位处理器） 但是安装了模拟的 qemu-aarch64 架构 能够实现 user-static 模式

表 1: 硬件与环境信息

2 问题描述——NTT 算法简介

我们继续回顾一下上次实验提到的傅里叶变换相关知识。

2.1 离散傅里叶变换

此处我们先引入离散傅里叶变换。工业界中，尤其是与时序数据相关的，具有周期性的数据，需要进行卷积（convolution）计算。

离散傅里叶变换（DFT）将有限长度的离散信号转换到频域的一种数学变换。对于一个长度为 N 的离散序列 $x[n]$，$n = 0, 1, , N{-}1$，其离散傅里叶变换 $X[k]$ 定义为：

$$\mathbf{X}[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j(2\pi/N)kn}, \quad (0 \le k \le N-1) \tag{1}$$

DFT 将时域信号 $x[n]$ 转换为频域信号 $X[k]$，其中 $e^{-j(2\pi/N)kn}$ 是单位复根，表示信号在不同频率上的投影。

在这里我们可以发现，多项式的系数被写成了复数的形式，算法复杂度是 $O(n^2)$ 的形式。

2.2 快速傅里叶变换

由上所述，这样的复杂度是无法接受的，所以我们引入快速傅里叶变换（fast fourier transform, FFT），通过多次迭代递归，划分为递归子问题实现算法优化，时间复杂度来到了 $O(n \log n)$，实现了优化。设 $x = \left[x_0, x_1, \cdots, x_{n-1}\right]^{\mathrm{T}}$ 是时域输入向量，设单位根 $\omega_n = e^{-2\pi i/n}$，则 DFT 可以表示为矩阵乘法：

$$X = F_n \cdot x \tag{2}$$

其中 DFT 矩阵 F_n 定义为：

$$F_n = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \tag{3}$$

因此，输出频域向量为：

$$X_k = \sum_{j=0}^{n-1} x_j \cdot \omega_n^{jk}, \quad k = 0, 1, \ldots, n-1 \tag{4}$$

其中 $\omega_n = e^{-2\pi i/n}$ 是 n 阶单位根。

2.3 数论变换

但是新的问题：复数需要引入虚数 i，在计算过程中涉及到了许多高精度浮点数计算（每个复数系数的实部和虚部是一个正弦及余弦函数，π 的原因导致），在多次重复的情况下误差累加会导致计算错误。最浅显的一点，对于复数需要额外的存储结构（C++ 中需要手搓结构体或者调用 complex 库），**浪费存储空间和运行时间**。这里我们从微观的浮点数视角跳出，看宏观的大数视角。

我们想到引入离散数学中关于同构和模的概念，利用原根和欧拉函数的性质，在一个周期内相反（余数相加等于模数）的两个数可以在单位圆上标记的特点，我们将圆 n 等分，利用同构实现和虚数根 ω^k 相同的效果。自然我们引入了 NTT 的概念。**数论变换**（number-theoretic transform, NTT）是离散傅里叶变换（DFT）在数论基础上的实现；快速数论变换（fast number-theoretic transform, FNTT）是快速傅里叶变换（FFT）在数论基础上的实现。

$$\mathbf{X}[k] = \sum_{n=0}^{N-1} x[n] \cdot \omega^{nk} \mod p, \ \ (k = 0, 1, \ldots, N-1) \tag{5}$$

其中，p 是一个素数，ω 是模 p 意义下的 N 次原根，满足 $\omega^N \equiv 1 \mod p$ 且 $\omega^{N/d} \not\equiv 1 \mod p$（对于 $d < N$ 的所有因子）。

数论变换是一种计算卷积的快速算法。主要应用在于计算多项式乘法，给定两个多项式 $A(x) = \sum_i a_i x^i$ 和 $B(x) = \sum_i b_i x^i$，其乘法结果 $C(x) = A(x) \cdot B(x)$ 的系数可以通过卷积计算：

$$c_k = \sum_{i+j=k} a_i b_j \tag{6}$$

使用 NTT：

1. 将 A 和 B 的系数向量进行 NTT，得到频域表示 A' 和 B'。
2. 计算点值乘法：$C'[k] = A'[k] \cdot B'[k] \bmod p$。
3. 对 C' 进行逆 NTT，得到卷积结果 c_k。

模数 p 限制了系数范围，需确保结果不超过 p。若结果超出，可使用多模数 NTT 结合中国剩余定理恢复。NTT 解决的是多项式乘法带模数的情况，可以说有些受模数的限制，数也比较大。目前最常见的模数是 998244353。

下面的算法分析，这里由于上次 SIMD 实验完成了朴素算法的分析，这里不再赘述，我们直接对 Pthread 和 OpenMP 进行分析。

3 实验设计：Pthread 的设计

对于 Pthread 来说，设计程序的核心在于设计控制句柄，选择合适的部分进行并行化操作。下面我们来详细说说这一部分，首先是一些要注意的事项：

1. **任务划分**：确保工作量均匀分配，避免某些线程过载影响性能。
2. **同步开销**：线程间需要适当同步以避免数据竞争，但需要注意同步可能带来的性能损失。
3. **缓存效率**：多线程访问共享数据可能导致缓存失效，影响性能。
4. **负载均衡**：不同线程的工作量需平衡，以充分利用多核资源。
5. **正确性验证**：并行化可能引入错误，需要与单线程版本对比验证。

基于 NTT 算法蝶形运算的阶段特性，Pthread 编程需遵循按阶段并行化的任务划分原则：通过 pthread_create 创建线程池时，应将计算任务按数组分块分配至各线程（如按阶段划分数据块），**确保不同线程处理的数据块无内存访问冲突**。这个过程需注意阶段间的依赖关系，可利用 pthread_barrier 等同步原语控制执行顺序，避免前一阶段未完成即启动后续计算。

线程同步策略：在蝶形运算的阶段切换点，**必须通过 pthread_barrier 等实现线程同步防止数据不一致**。但需避免过度同步（如频繁加锁），以免增加上下文切换开销。

数据访问优化：减少共享数据的全局访问，**优先使用 __thread 关键字声明线程本地存储（TLS）**，降低缓存失效概率。针对可能出现的伪共享问题（如多个线程频繁访问相邻缓存行），可通过 pthread_cacheline_align 等机制对数据结构进行对齐或填充。

根据目标硬件的物理核心数动态调整线程数（通常不超过核心数），可尝试通过 pthread_attr_setaffinity_np **绑定线程至指定核心，避免 CPU 调度引起的负载失衡**。对于不同阶段的计算复杂度差异，可采用动态任务分配（如使用 pthread_workqueue），实现运行时的负载均衡。

可以的进一步尝试：并行化实现需**重点防范死锁、数据竞争等并发错误**。利用 pthread_mutex_timedlock 等接口避免死锁，通过 helgrind 等工具检测竞态条件；每次并行计算后，需与单线程版本的结果进行比对（如通过 memcmp 校验中间结果），确保算法正确性；对共享数据的修改操作必须使用 pthread_mutex-_lock 加锁保护，防止指令重排序导致的数据错误。

基于此我们设计了两种方式：

3.1 朴素多线程优化

对于 NTT 的朴素多线程优化，我们的要点在于插入句柄的位置。

```
for(int mid = 1; mid < limit; mid <<= 1) {
    for(int j = 0; j < limit; j += (mid << 1)) {
        int w = 1;// 旋转因子
        for(int k = 0; k < mid; k++, w = w  Wn) {
```

```
            // 对这层循环进行优化
            // 运算主体
        }
    }
}
```

由于第二三层循环相当于遍历了一遍多项式数组，显然可以对第三层循环进行多线程优化，类似高斯消元，因此代码实现较为简单，只需要注意线程同步正确即可正确实现。

下面是我们仿照这一过程设计的程序（仅列出蝶形变换）：

```
// 逐层进行蝶形运算
for (int len = 2; len <= n; len <<= 1) {
    // 旋转因子的定义+线程创造
    __int128 wlen = mod_pow(g, (p - 1) / len, p);
    if (invert) wlen = mod_inv(wlen, p);

    pthread_t threads[THREADS];
    NTTParams params[THREADS];

    // 计算每个线程的任务
    int num_blocks = n / len; // 总共的蝶形运算单元数
    int blocks_per_thread = (num_blocks + THREADS - 1) / THREADS; //
        每个线程的块数（向上取整）
    int block_size = len; // 每个蝶形运算单元的大小

    for (int t = 0; t < THREADS; ++t) {
        int start_block = t * blocks_per_thread;
        int thread_blocks = std::min(blocks_per_thread, num_blocks - start_block);
        if (thread_blocks <= 0) {
            params[t] = {a, len, p, wlen, 0, 0, block_size};
        } else {
            params[t] = {a, len, p, wlen, start_block * len, thread_blocks,
                block_size};
        }
        // 这里的ntt_layer_thread对应了前面示例中的最后一层循环
        pthread_create(&threads[t], nullptr, ntt_layer_thread, &params[t]);
    }

    for (int t = 0; t < THREADS; ++t)
        pthread_join(threads[t], nullptr);
}
```

前面定义的 Thread 变量将整个计算过程分成了大的部分，按照线程数进行。

下一步将当前层的蝶形运算单元（共 n / len 个）分配给多个线程（THREADS 个）。每个线程处理若干个蝶形单元（blocks_per_thread），每个单元大小为 len。通过多线程并行化了上述过程，将 num_blocks 个单元分配给多个线程处理，相当于并行化了示例中的 j 循环。

优化效果如下：

1. **负载均衡：** blocks_per_thread 确保任务均匀分配，最后一个线程可能处理较少的块。这是一种

常见的负载均衡策略，但当 num_blocks 远小于 THREADS 时，部分线程可能空闲，可以考虑动态调整线程数。

2. **线程开销:** 创建和销毁线程（pthread_create 和 pthread_join）有一定开销。对于小规模数据（n 较小），多线程的开销可能超过并行化的收益。在后面的对比实验中能够看到这一点。

继续为每个线程分配任务参数（NTTParams），pthread_create 创建线程，ntt_layer_thread 执行蝶形运算，并通过 pthread_join 等待所有线程完成。将单个蝶形单元内的（处理 mid 对元素）这些计算外包给 ntt_layer_thread，实现了对每个块内的 len/2 对元素进行蝶形运算的相似逻辑。

3.2 多分多线程优化

由于已经实现了 CRT 多线程，所以这一部分仅把代码放在 Github 上面，不再赘述。核心点在于

3.3 CRT 多线程优化

CRT（中国剩余定理）的多模数 NTT 算法，以及使用 Pthread 进行并行化加速的优化版本。主要目标是支持大模数输入（超过 32 位），并通过多线程优化提高计算效率。

任意模数 NTT 与 CRT 标准 NTT 要求模数满足特定形式（如 $p = a \times 2^k + 1$），且需要存在原根。对于任意模数，可以使用多个小模数进行 NTT 计算，然后通过中国剩余定理 (CRT) 合并结果。

CRT 原理 若有互质的模数 $m_1, m_2, ..., m_k$，对于同余方程组：

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \quad \cdots \\ x \equiv a_k \pmod{m_k} \end{cases} \tag{7}$$

存在唯一解 $x \equiv \sum_{i=1}^{k} a_i \cdot M_i \cdot M_i^{-1} \pmod{M}$，其中：
$M = m_1 \times m_2 \times ... \times m_k$ $M_i = M/m_i$ M_i^{-1} 是 M_i 在模 m_i 下的逆元。
接下来，我们开始讨论一下具体的优化过程。

Pthread 并行化优化 多线程优化主要分为两个层次：
模数级并行：每个小模数下的 NTT 计算分配给独立线程
NTT 内部并行：蝶形运算阶段使用多线程并行处理

Algorithm 1 基于 CRT 的并行多项式乘法（Pthread 优化）

Input: 多项式 $a[0 \dots n-1]$、$b[0 \dots n-1]$，结果数组 $r[0 \dots 2n-2]$，长度 n，模数 p
Output: 卷积结果 $r \equiv a * b \pmod{p}$

```
function PolyMultiply(a, b, r, n, p)
    初始化线程池，包含 t 个线程                        ▷ 重复使用线程以减少创建开销
    num_mods ← 若 p > 2^32 则 5 否则 3
    moduli ← {998244353, 754974721, 167772161, 469762049, 1004535809}[: num_mods]
    预计算 CRT 常数：M ← ∏moduli，M_i ← M/moduli[i]，M_i^{-1} ← mod_inverse(M_i, moduli[i])
```

```
    分配 results[num_mods][2n-1]                                    ▷ 存储各模数下的结果
    for i = 0 to num_mods−1 do                                      ▷ 模数级并行
        提交任务到线程池：PolyMultiplyMod(a, b, results[i], n, moduli[i])
    end for
    等待所有模数任务完成
    result_size ← 2n − 1
    t_crt ← min(t, ⌈result_size/1024⌉)                              ▷ 动态调整 CRT 线程数
    chunk_size ← ⌈result_size/t_crt⌉
    for tid = 0 to t_crt − 1 do                                     ▷ CRT 合并并行
        start ← tid × chunk_size
        end ← min(start + chunk_size, result_size)
        if start < result_size then
            提交任务到线程池：CRTMerge(results, r, start, end, p, num_mods)
        end if
    end for
    等待所有 CRT 任务完成
    释放 results
end function
function PolyMultiplyMod(a, b, r, n, m)
    ℓ ← 2^⌈log_2(2n)⌉                                               ▷ 填充到 2 的幂
    分配 ta[ℓ], tb[ℓ] ← 0
    将 a[0 . . . n − 1]、b[0 . . . n − 1] 复制到 ta、tb，模 m
    ParallelNTT(ta, ℓ, m, false, t)                                 ▷ 前向 NTT
    ParallelNTT(tb, ℓ, m, false, t)
    for i = 0 to ℓ − 1 do
        ta[i] ← (ta[i] × tb[i]) mod m                               ▷ 点值乘法
    end for
    ParallelNTT(ta, ℓ, m, true, t)                                  ▷ 逆 NTT
    将 ta[0 . . . 2n-2] 复制到 r
end function
function CRTMerge(results, r, start, end, p, num_mods)
    M ← crt_consts[num_mods].M
    for i = start to end − 1 do
        x ← 0
        for j = 0 to num_mods−1 do
            M_j ← crt_consts[num_mods].Mi[j]
            M_j^{-1} ← crt_consts[num_mods].Mi_inv[j]
            x ← (x + (results[j][i] × M_j × M_j^{-1}) mod M) mod M
        end for
        r[i] ← p ? (x mod p) : x
    end for
end function
```

对于模数的选取，我们选择了五模数的方式进行代码的实现，通过五个模数分别计算再把结果合并，从而达到我们的目标。

```
// 这些模数都满足形如 a * 2^k + 1 的形式
const __int128 MOD1 = 998244353;       // 2^23 * 119 + 1
const __int128 MOD2 = 754974721;       // 2^24 * 45 + 1
const __int128 MOD3 = 167772161;       // 2^25 * 5 + 1
const __int128 MOD4 = 469762049;       // 2^26 * 7 + 1
const __int128 MOD5 = 1004535809;      // 2^21 * 479 + 1

// 对应的原根
const int PRIMITIVE_ROOT1 = 3;
const int PRIMITIVE_ROOT2 = 11;
const int PRIMITIVE_ROOT3 = 3;
const int PRIMITIVE_ROOT4 = 3;
const int PRIMITIVE_ROOT5 = 3;
```

```
// 中国剩余定理合并结果
__int128 crt(const std::vector<__int128>& remainders, const std::vector<__int128>&
    moduli, __int128 target_mod) {
    __int128 result = 0;
    __int128 M = 1;

    // 计算所有模数的乘积
    for (const auto& mod : moduli) {
        M *= mod;
    }

    // 应用中国剩余定理
    for (size_t i = 0; i < remainders.size(); ++i) {
        __int128 Mi = M / moduli[i];
        __int128 Mi_inv = mod_inverse(Mi, moduli[i]);

        // 注意这里的计算顺序，避免中间结果溢出
        __int128 term = remainders[i];
        term = (term * Mi) % M;
        term = (term * Mi_inv) % M;
        result = (result + term) % M;
    }

    // 如果结果需要对目标模数取模
    if (target_mod > 0) {
        // 处理结果可能超过目标模数一半的情况
        if (result >= M / 2) {
            result = result - M;
        }

        // 对目标模数取模，处理负数情况
        result = ((result % target_mod) + target_mod) % target_mod;
```

```
    }

    return result;
}
```

上面的代码实现了 CRT 的合并模数部分。我们开了较大的数据类型，让模数的表示范围变大，最后能表达的模数也会非常大。在一定程度上可以实现硬件指标的优化。

4 实验设计：OpenMP 的设计

对于这一部分的设计，我们可以在朴素算法中加入线程控制，也可以在实现了 NTT 的朴素算法下添加。这里为了持续优化，我们在实现了 NTT 的朴素算法下面添加。

4.1 蝶形变换中的并行化

```
for (int len = 2; len <= n; len <<= 1) {
    int half = len >> 1;
    int wlen = mod_pow(w, (p - 1) / len, p);
    #pragma omp parallel for
    for (int i = 0; i < n; i += len) {
        int w_now = 1;
        for (int j = 0; j < half; ++j) {
            int u = a[i + j];
            int v = (long long)a[i + j + half] * w_now % p;
            a[i + j] = (u + v) % p;
            a[i + j + half] = (u - v + p) % p;
            w_now = (long long)w_now * wlen % p;
        }
    }
}
```

我们不难发现，外层循环中的 for (int i = 0; i < n; i += len) 将数组分成了多个长度为 len 的独立块。每个块的计算（内层循环）仅仅依赖于这个块内的数据，**不同块之间的数据没有数据依赖**。这种独立性使得这些块的蝶形运算可以同时执行，无需担心数据竞争或者同步。

使用 #pragma omp parallel for，OpenMP 会将外层循环的迭代分配给多个线程，每个线程处理不同的 i 值范围，从而并行计算多个块的蝶形运算。当 n 较大时，这种并行化可以显著减少 NTT 的计算时间。

4.2 输入的并行化

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    ta[i] = a[i];
    tb[i] = b[i];
}
```

该循环将输入数组 a 和 b 的值复制到临时数组 ta 和 tb 中（两个多项式的点值表示）。**每个迭代（即每个 i）的赋值操作是完全独立的**，不涉及共享数据的修改，也不存在数据竞争。这种独立性非常适合并行化处理.

通过 #pragma omp parallel for，OpenMP 将循环迭代分配给多个线程，多个线程可以同时执行赋值操作。尤其当 n 较大时，并行化能够加快数据复制的速度，从而减少初始化阶段的开销。

4.3 点值乘法的并行化

```
#pragma omp parallel for
for (int i = 0; i < m; ++i) {
    ta[i] = (long long)ta[i] * tb[i] % p;
}
```

点值乘法是一个逐元素操作，每个 i 的计算（ta[i] * tb[i] % p）仅依赖于 ta[i] 和 tb[i]，与其他位置的计算无关。这种独立性允许所有乘法操作并行执行。

使用 #pragma omp parallel for，OpenMP 将循环分成多个部分，分配给不同线程并行计算。当 m（数组长度）较大时，这种并行化可以显著缩短点值乘法的时间。

4.4 归一化操作的并行化

```
#pragma omp parallel for
for (int i = 0; i < 2 * n - 1; ++i) {
    ab[i] = (long long)ta[i] * inv_m % p;
}
```

在逆 NTT 变换后，需要对结果进行归一化处理，即将每个元素乘以模逆 inv_m 并取模。**每个 i 的归一化操作（ta[i] * inv_m % p）是独立的**，不依赖于其他位置的计算结果。这种独立性适合并行执行。

通过 #pragma omp parallel for，OpenMP 将循环分配给多个线程并行处理。对于较大的 n，并行化可以加速归一化过程，从而更快生成最终结果。

5 程序性能分析

这次实验，我们采用下面的编译方式：

```
g++ main.cc -o main -O2 -fopenmp -lpthread -std=c++11
bash test.sh 2/3 1 线程数
```

其中 2 代表 Pthread，3 代表 OpenMP。线程数是我们选择的线程数量，默认设置为 8。

5.1 Pthread 程序之间的比较

接下来，我们首先看看 Pthread 下面的两种算法的优化（由于我们实现了 CRT 算法的优化，所以不考虑将四分算法放进来对比）

如上表2所示，进行优化之后的算法当规模增大的时候，

算法	n	p	平均延迟 (us)	加速比
NTT 实现	4	7340033	0.00425	1.000
	131072	7340033	83.9529	1.000
	131072	104857601	88.6243	1.000
	131072	469762049	91.4264	1.000
NTT_Pthread（优化）	4	7340033	0.844011	0.00503
	131072	7340033	78.1798	1.074
	131072	104857601	81.0444	1.093
	131072	469762049	78.2513	1.168
Ntt_Pthread+CRT	4	7340033	0.35111	0.0121
	131072	7340033	626.234	0.134
	131072	104857601	629.878	0.141
	131072	469762049	630.586	0.145

表 2: 多项式乘法算法性能比较（含加速比）

- **NTT 实现（基准）**:
  - 作为基准算法，加速比固定为 1.000。
  - 对于小规模输入 ($n = 4, p = 7340033$)，平均延迟极低（0.004 25 ţs）。
  - 对于大规模输入 ($n = 131072$)，随着模数 p 增大（从 7340033 到 469762049），平均延迟从 83.9529 ţs 增至 91.4264 ţs，表明模数对性能影响较小。
- **NTT_Pthread 优化**:
  - 在小规模输入 ($n = 4, p = 7340033$) 下，平均延迟为 0.844 011 ţs，性能较基准大幅下降（加速比 0.00503）。
  - 在大规模输入 ($n = 131072$) 下，平均延迟在 78.1798 ţs 至 78.2513 ţs 之间，优于基准（加速比 1.074–1.168），显示出优化效果，尤其在较大模数 ($p = 469762049$) 下加速比最高 (1.168)。
- **NTT_Pthread+CRT**:
  - 在小规模输入 ($n = 4, p = 7340033$) 下，平均延迟为 0.351 11 ţs，性能仍逊于基准（加速比 0.0121）。
  - 在大规模输入 ($n = 131072$) 下，平均延迟显著增加（626.234 ţs–630.586 ţs），性能远低于基准（加速比 0.134–0.145），表明 CRT 优化在大规模计算中引入了较高开销。

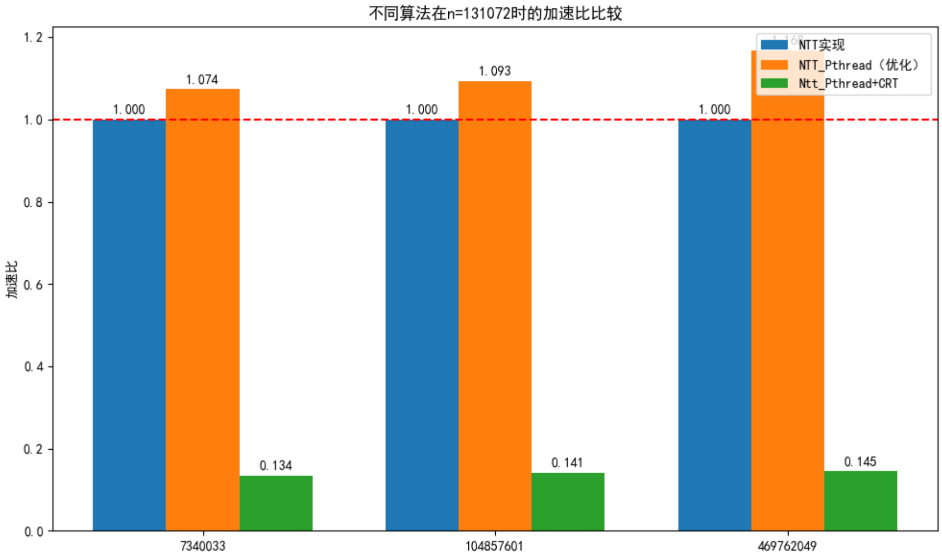

图 5.1: 加速比可视化

上面的可视化图像也可以帮助我们明白这一过程（这里没有考虑 n=4 的时候，这种情况时间极短，对硬件的影响意义不大）。

对线程数的讨论 这里还要补充一点关于线程数对我们程序的影响要素，这里我们选取 Pthread 普通加速的程序进行讨论。由于 CRT 我们根据模数的个数指定了线程数，这里不再做相关的实验了，只有后面和其他算法的同步验证（但基准指的是以 1 线程数为基准进行比较）：

多项式长度	模数	线程数	平均延迟 (us)	加速比	备注
$n = 4$	$p = 7340033$	1	1.159	–	单基准
		2	0.665	1.741	–
		4	0.830	1.395	–
		8	0.831	1.394	–
$n = 131072$	$p = 7340033$	1	188.611	–	单基准
		2	123.616	1.526	–
		4	82.459	2.287	–
		8	85.239	2.213	–
$n = 131072$	$p = 104857601$	1	188.275	–	单基准
		2	129.308	1.456	–
		4	89.313	2.108	–
		8	83.862	2.245	–
$n = 131072$	$p = 469762049$	1	189.835	–	单基准
		2	119.651	1.587	–
		4	100.845	1.883	–
		8	81.723	2.322	–

表 3: Pthread 多线程 NTT 性能比较（排除大模数的错误结果）

上面的表3能够反映出：**线程数越多，加速效果越明显**，尤其是对于大规模的多项式以及大规模的模数效果更佳。并结合下面的图5.2印证了这一观点的成立。

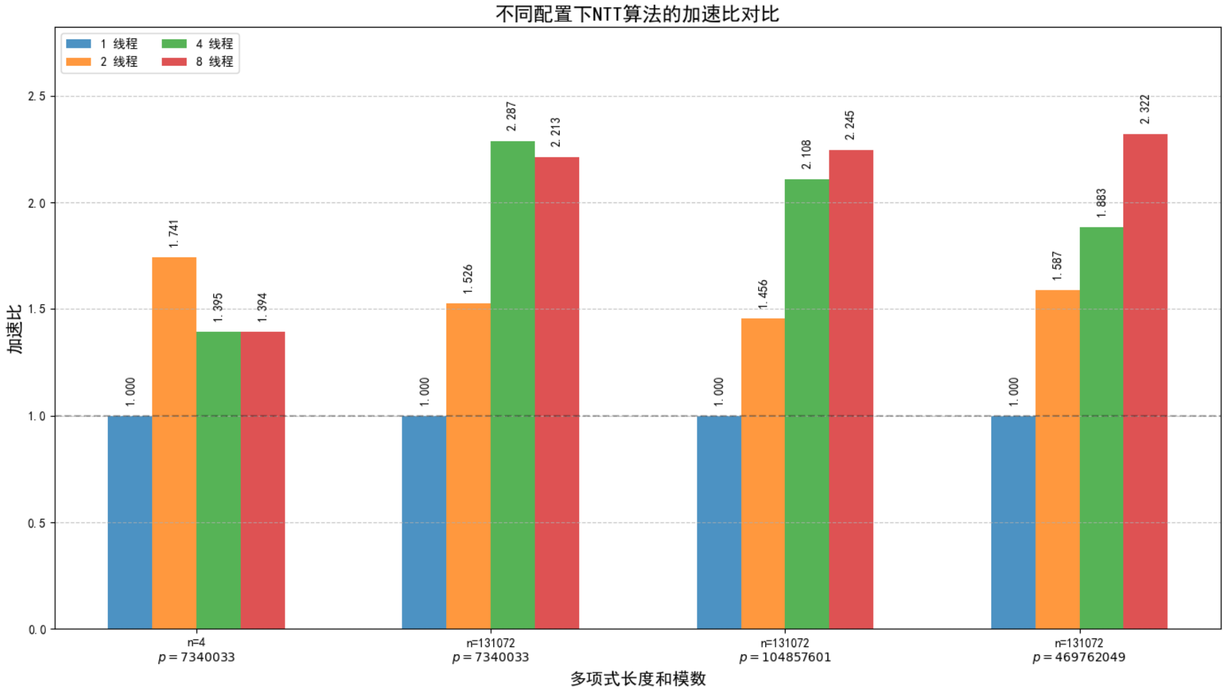

图 5.2: 加速比可视化

总体结论 由上面的图5.1可以看到：

- NTT_Pthread 优化在大规模输入下表现最佳，平均延迟低于基准，且加速比随模数增大而提升，适合高性能场景。
- NTT_Pthread+CRT 在小规模输入下延迟较低，但在大规模输入下性能显著下降，加速比远低于 1，表明不适合并行化之后进行大规模多项式乘法。
- 基准 NTT 实现虽在小规模输入下极高效，但在大规模输入下被 NTT_Pthread 优化超越。

条件	说明
小规模 $n \leq 1024$	启动线程反而会拖慢速度
中等规模 $n = 2^{15} \sim 2^{17}$	**非常适合多线程**，可提速 1.5~3 倍
大规模 $n \geq 2^{18}$	推荐使用线程池或 OpenMP

表 4: 不同输入规模下的多线程适用性分析

显然发现我们的测试样例出现了大规模数据，使用多线程可能并不能很好地带动计算，所以推荐使用线程池或者 OpenMP 进行操作，下面便引出了我们的测试信息。

5.2 Pthread 和 OpenMP 程序之间的比较

这里来说 OpenMP 的优化方式较少，所以我们就采用一种方式（对朴素 NTT 进行 OpenMP 的指令插入），然后与前面的 Pthread 编程（只选择了 Pthread 优化的程序）进行结合进行对比分析。

我们能够看到图5所示的数据：

- **NTT 实现（基准）**：

算法	n	p	平均延迟（us）	加速比
NTT 实现	4	7340033	0.00425	1.000
	131072	7340033	83.9529	1.000
	131072	104857601	88.6243	1.000
	131072	469762049	91.4264	1.000
NTT_Pthread（优化）	4	7340033	0.844011	0.00503
	131072	7340033	78.1798	1.0739
	131072	104857601	81.0444	1.0936
	131072	469762049	78.2513	1.1681
NTT_OpenMP	4	7340033	0.47124	0.00902
	131072	7340033	22.6913	3.7003
	131072	104857601	23.8526	3.7166
	131072	469762049	24.49	3.7325

表 5: 多项式乘法算法性能比较（含加速比）

- 作为基准算法，加速比固定为 1.000。
- 对于小规模输入 ($n = 4, p = 7340033$)，平均延迟极低，仅为 0.004 25 ţs。
- 对于大规模输入 ($n = 131072$)，随着模数 p 从 7340033 增加到 469762049，平均延迟从 83.9529 ţs 增至 91.4264 ţs，表明模数增大对性能影响较小。

- **NTT_Pthread 优化**：
  - 在小规模输入 ($n = 4, p = 7340033$) 下，平均延迟为 0.844 011 ţs，性能较基准显著下降（加速比 0.00503）。
  - 在大规模输入 ($n = 131072$) 下，平均延迟在 78.1798 ţs 至 78.2513 ţs 之间，略优于基准（加速比 1.0739–1.1681），显示出优化效果，尤其在较大模数 ($p = 469762049$) 下加速比最高（1.1681）。

- **NTT_OpenMP**：
  - 在小规模输入 ($n = 4, p = 7340033$) 下，平均延迟为 0.471 24 ţs，性能低于基准（加速比 0.00902），但优于 NTT_Pthread。
  - 在大规模输入 ($n = 131072$) 下，平均延迟显著降低（22.6913 ţs–24.49 ţs），远优于基准和 NTT_Pthread，加速比高达 3.7003–3.7325，展现出卓越的并行优化效果，尤其在模数增大时性能稳定。

下面的图5.3也反映出我们程序的相关情况（仍然没有考虑 n=4 的时候，这种情况时间极短，对硬件的影响意义不大）

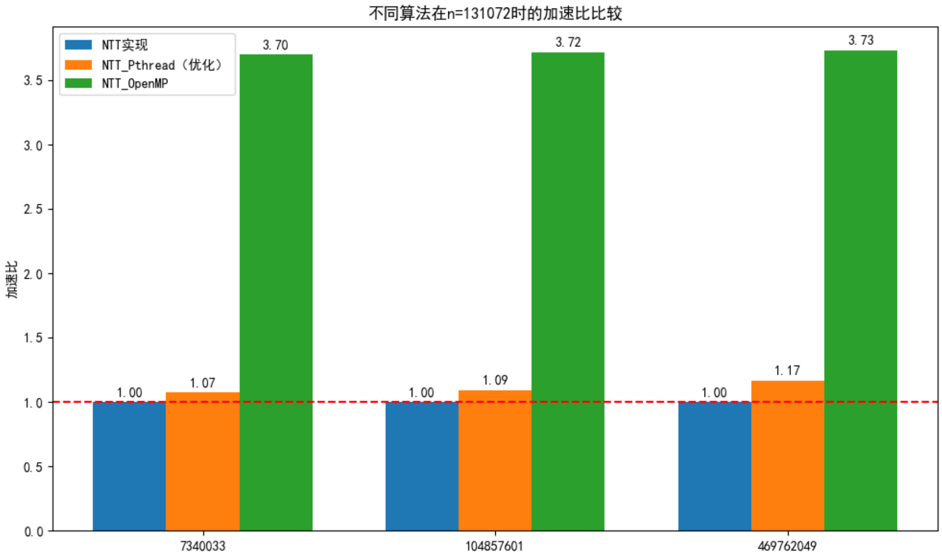

图 5.3: 加速比可视化

总体结论 由上面的图5.3可以看到：

- NTT_OpenMP 在大规模输入下表现最佳，平均延迟远低于基准和 NTT_Pthread，加速比高达 3.7 倍以上，适合高性能大规模多项式乘法场景。
- NTT_Pthread 优化在大规模输入下略优于基准，加速比略超 1，但在小规模输入下性能较差。
- 基准 NTT 实现虽在小规模输入下高效，但在大规模输入下被 NTT_OpenMP 和 NTT_Pthread 优化显著超越，表明并行优化对大规模计算至关重要。

6 Profiling

这次实验，我们继续采用 Perf 进行事件采样，获得相应数据进行分析。

Perf 的测试

```
perf record -g -e cpu-clock,cycles,instructions,cache-references,cache-misses,
L1-dcache-loads,L1-dcache-load-misses,LLC-loads,LLC-load-misses ./main

perf report > 文件名.txt
```

我们使用如上的指令进行性能分析，stat 的方式可以直接查看相应的数据，record 可以获得更为详细的数据记录。使用获取的数据从上述维度探究程序性能的影响（这里不提供普通算法）。

6.1 数据的总览

由上表可以知道，对于需要高效利用 CPU 和缓存资源的场景，**OpenMP 优化是最佳选择**。但是我们的 OpenMP 在某种程度上没有朴素的 NTT 效果好，可能是由于线程问题导致。

进一步介绍并详细说明，**pthread 优化**在 CPU 时钟周期上表现良好，但可以通过优化 ntt_layer_thread 函数和改进内存访问模式来进一步提升性能。

指标	NTT 朴素	pthread	fourDivide	CRT	OpenMP
CPU 时钟周期 ($\times 10^9$)	4.328	13.065	90.764	62.327	**11.974**
指令数量 ($\times 10^9$)	2.351	2.416	31.926	31.691	5.969
每周期指令数	1.54	1.64	2.01	2.04	**2.27**
每指令缓存引用	0.300	**0.256**	0.089	0.089	0.222
缓存未命中率 (%)	0.51	0.87	0.51	0.46	**0.27**
L1 数据缓存未命中率 (%)	0.51	0.88	0.51	0.47	**0.28**
最后级缓存未命中率 (%)	16.08	51.28	38.79	36.95	**23.58**

表 6: 优化方法间的硬件性能对比分析（朴素只作为 baseline，不参与比较）

fourDivide 四分优化和 **CRT 优化**可以通过优化 poly_multiply_single_mod 和 ___modti3 函数来提升性能，特别是减少这些函数执行的指令数量。

所有优化方法都可以通过改进内存访问模式来降低缓存未命中率，特别是 pthread 优化，其最后级缓存未命中率较高。这也是未来我们要改进的方向。

下面是对于上述数据的分析：

1. OpenMP 优化在大多数性能指标上表现最佳，特别是在 CPU 时钟周期、缓存未命中率和最后级缓存未命中率等关键指标上。其优势在于高效的内存访问模式和良好的指令级并行性。

2. pthread 优化在 CPU 时钟周期上表现良好，仅略高于 OpenMP，但在缓存未命中率上表现较差。其主要瓶颈在于 ntt_layer_thread 函数。

3. fourDivide 优化和 CRT 优化在性能上表现相似，都执行了大量的指令，且主要瓶颈都在 poly_multiply_single_mod 和 ___modti3 函数。它们在缓存未命中率上表现中等，但在 CPU 时钟周期上消耗较多。

4. 从指令执行效率和缓存利用率来看，四种优化方法的排名为：**OpenMP > CRT > fourDivide > pthread**。

6.2 更进一步的解释

对线程数的讨论 首先，我们继续分析硬件上不同线程数对程序性能的影响。如下表7我们发现 4 线程数对于程序来说是硬件友好的，相关指标都达到了较好的标准。但是这不能说明 4 线程就是最好的，我们后续还可以继续实验，探究不同线程数对硬件性能的分析。

指标	1 线程	2 线程	4 线程	8 线程
每周期指令数（IPC）	2.28	2.16	**1.67**	2.15
每指令缓存引用	0.308	0.309	0.301	**0.321**
缓存未命中率（%）	0.111	0.261	**0.428**	0.209
L1 数据缓存未命中率（%）	0.107	0.398	**0.440**	0.203
LLC 未命中率（%）	41.24	17.63	21.87	**116.91**

表 7: 不同线程数下衍生性能指标

这个热力图6.4能够让我们清晰看到，对应线程中哪个函数的贡献最大，对于为命中率等方面，这也是我们后续涉及算法和代码的优化方向。详细解说如下：

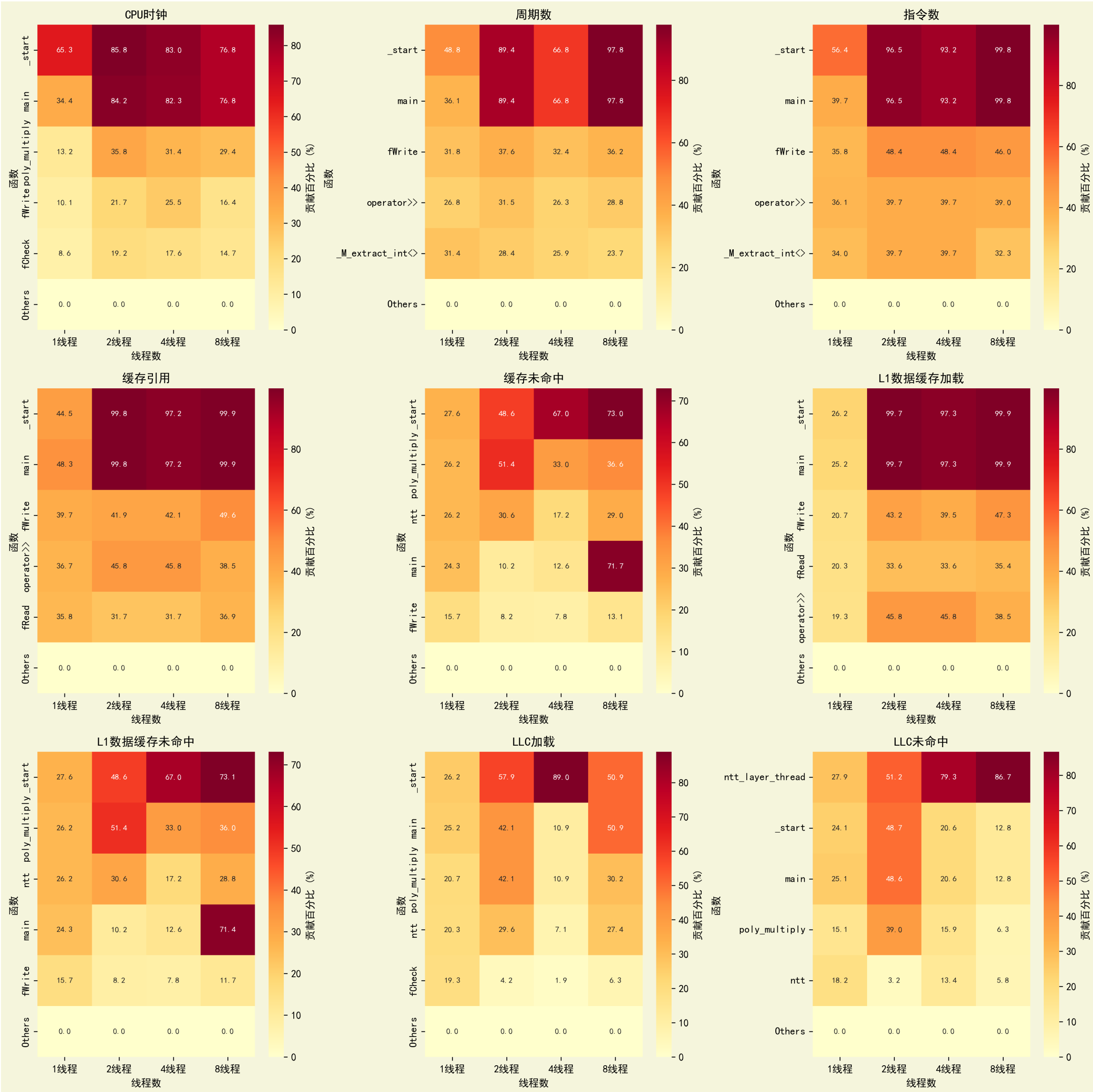

图 6.4: 不同线程对硬件事件的影响

- **指标分析**
  - **CPU 时钟**：线程数增加时，线程管理函数（如_start 和main）的贡献显著上升，尤其在 2 线程及以上，显示线程初始化成本的激增。单线程时，计算核心函数（如ntt_layer_thread）占主导，而多线程下 I/O 和输入操作（如poly_multiply 和std::istream::operator>>）的重要性逐渐显现。
  - **周期数**：多线程环境下，_start 和main 的周期贡献大幅提升，特别是在 8 线程接近饱和，表明线程管理开销成为主要瓶颈。输入操作在单线程时占主导，但随线程数增加其相对影响减弱。
  - **指令数**：随着线程数增加，_start 和main 几乎完全主导指令执行，特别是在 8 线程接近 100%，反映线程管理对指令分配的绝对控制。I/O 函数在高线程数下仍有一定贡献。
  - **缓存引用**：线程数增加，_start 和main 的缓存引用贡献显著上升，I/O 操作保持较高比例，显示缓存访问主要集中于线程管理和文件操作，而非计算核心。

  - **缓存未命中**：中高线程数下，`ntt_layer_thread` 的未命中率显著上升，特别是在 4 线程达到峰值，表明多线程环境下缓存竞争加剧。8 线程时，`poly_multiply` 和`ntt` 的影响增强。
  - **L1 数据缓存加载**：线程管理函数在多线程下主导 L1 缓存加载，I/O 函数保持稳定贡献，显示缓存加载主要服务于线程管理和数据写入。
  - **L1 数据缓存未命中**：与缓存未命中类似，`ntt_layer_thread` 在 4 线程时贡献峰值，随后 8 线程时`_start` 反弹，表明缓存未命中随线程数增加呈现波动。
  - **LLC 加载**：`ntt_layer_thread` 在 4 线程时达到最高贡献，随后 8 线程时`_start` 和`main` 反弹，显示 LLC 加载在中等线程数下集中于计算任务。
  - **LLC 未命中**：`ntt_layer_thread` 随线程数增加显著上升，特别是在 8 线程接近 90%，反映高线程数下 LLC 缓存的严重竞争。

- **总体趋势与洞察**
  - **线程管理开销**：`_start` 和`main` 在多线程下贡献急剧上升，尤其在指令数和缓存引用中接近 100%，显示线程初始化和管理成本的激增。
  - **计算与 I/O 平衡**：`ntt_layer_thread` 和`poly_multiply` 在单线程和缓存相关指标中占主导，但多线程下`fWrite` 和输入操作的重要性增强。
  - **缓存压力**：`ntt_layer_thread` 在缓存未命中和 LLC 指标中随线程数增加显著，特别是在 4-8 线程，表明多线程下缓存竞争加剧。
  - **异常点**：8 线程的 LLC 未命中率接近 90%，可能需要验证数据准确性或优化缓存使用。

- **优化建议**
  1. 针对 4-8 线程的高缓存未命中率，优化`ntt_layer_thread` 和`poly_multiply` 的缓存访问模式，减少竞争。
  2. 检查 8 线程的 LLC 加载和未命中数据，确保无异常。
  3. 有可能的话减少线程管理开销，例如优化`_start` 和`main` 的初始化逻辑。
  4. 进一步收集多样本数据，分析函数贡献的分布以支持更精确的优化。

结合 OpenMP 再谈加速 如下图6.5所示，这是我们总程序的一个情况。不难发现在缓存未命中率相关指标中，优化后的 OpenMP 实现相较于普通 pthread 实现显著提高了缓存命中率。缓存未命中（cache-misses）主要来源于程序本身而非外部因素，这有效提升了程序性能（占比反映对应关系而非绝对数量）。此外，OpenMP 在 CPU 指令效率上的表现也大幅优于其他方法。

通过对比普通 pthread 实现与优化后的 fourDivide、CRT 和 OpenMP 在不同性能指标下的表现，结果表明 OpenMP 优化显著提高了程序的执行效率。具体表现为：

1. **每周期指令数**：优化后的 OpenMP 实现达到 2.27，显著高于 pthread 的 1.64、fourDivide 的 2.01 和 CRT 的 2.04，表明 OpenMP 在指令级并行性上具有明显优势，提升了 CPU 的执行效率。
2. **每指令缓存引用**：pthread 实现以 0.256 领先，优于 OpenMP 的 0.222 以及 fourDivide 和 CRT 的 0.089，显示 pthread 在内存访问集中性上更强。然而，OpenMP 的值仍保持在较高水平，优于其他优化方法。

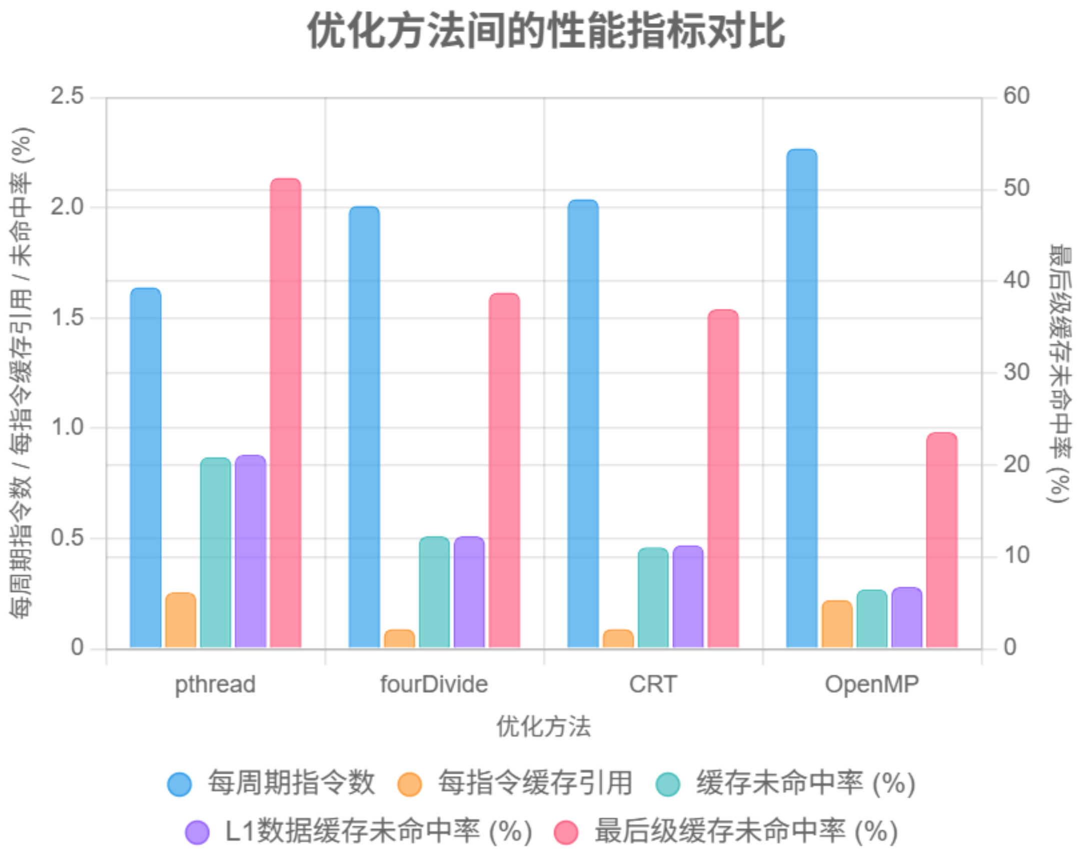

图 6.5: 程序运行总情况

3. **缓存未命中率**：OpenMP 实现的缓存未命中率最低，仅为 0.27 %，远低于 pthread 的 0.87 %、fourDivide 的 0.51 % 和 CRT 的 0.46 %。这表明 OpenMP 优化显著提升了缓存局部性，减少了不必要的缓存失效。
4. **L1 数据缓存未命中率**：OpenMP 在 L1 数据缓存未命中率上表现最佳，为 0.28 %，相比 pthread 的 0.88 %、fourDivide 的 0.51 % 和 CRT 的 0.47 %，进一步验证了其高效的内存访问模式。
5. **最后级缓存未命中率**：OpenMP 实现的最后级缓存未命中率最低，为 23.58 %，显著优于 pthread 的 51.28 %、fourDivide 的 38.79 % 和 CRT 的 36.95 %，显示出 OpenMP 在深层缓存利用上的卓越性能。

综上所述，OpenMP 优化极大地提升了程序性能，主要体现在更高的每周期指令数、更低的缓存未命中率以及优化的内存访问效率。特别是 OpenMP 在所有缓存相关指标上的表现最为突出，展示了并行优化在提升大规模计算性能方面的显著优势。

7 实验总结

实验的代码在我的 GitHub 网站上：我的 GitHub

7.1 本实验的概括总结

在使用 Pthread 和 OpenMP 进行算法优化时，对于各条指令的熟悉成为了我学习过程中的最大难点。经过一段时间的学习和钻研，我对该指令集有了一个初步的了解，进而一步一步实现自己的编程。

实验过程中，当我使用 Pthread 加速的过程中，我猛然发现有时候的优化效果是负的，一开始特别害怕，为什么并行化操作会带来反向的影响呢？后来查阅相关资料才知道，并行化操作有时候会有很大开销，这一部分会在我们的运行时间中有所体现。但是在硬件上依然是硬件友好的，能够节约资源，也有一种“时间换空间”的感觉。

虽然最后还是没有实现较大模数的计算，但是后面我会继续发掘更多新的算法来实现这一特点。

在实验过程中，我意识到这让我深刻体会到，OpenMP 优化的成功不仅依赖于指令级并行，还需要精准的实验设计和数据分析来捕捉性能提升的每一个细节，将多线程的语句安插在合适的地方，才能更好地利用这个强大的武器。

7.2 实验以外的总结

这次实验对我来说仍然是个极大的挑战。在事务繁忙的 5 月，我开始实验的时间相对较晚，时间紧任务重，在实验过程中也是遇到了一些困难。但是我通过一系列的学习开始慢慢熟悉这套过程，能够熟悉原理和细节上的要点，最后能较为不错的完成实验，对整个过程有了清晰且深刻的体会。这对于我以后的学习很有帮助。

最后，由衷感谢助教学长们的辛勤付出，他们对于我们实验的配置和实施给予了很大帮助。希望未来我能够好好学习这门课程，尽可能多地学会知识，将我所学的知识传递下去帮助更多小同学们！再次衷心感谢我们的老师和助教们的辛勤付出！

附录 A 实验的相关图像

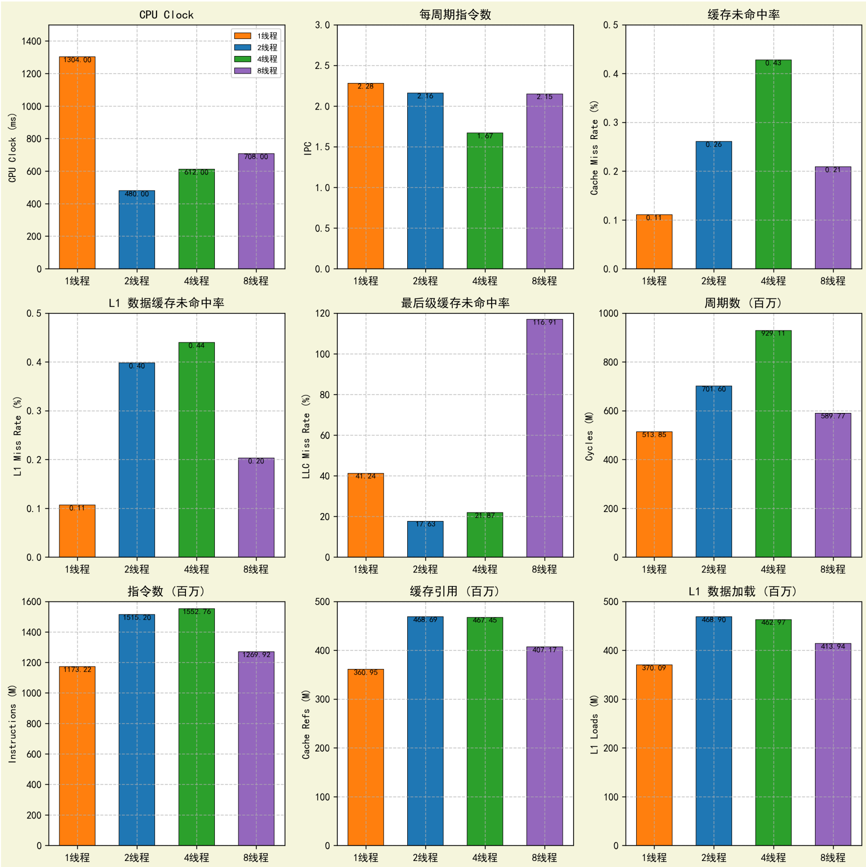

图 A.6: 不同线程对硬件事件的影响

附录 B 实验的详细表格

指标	朴素	pthread	fourDivide	CRT	OpenMP
cpu-clock:u	4327995672	13,065,320,268	90,763,909,236	62,327,271,006	11,973,988,026
cycles:u	1525948650	1,471,816,286	15,887,018,796	15,508,354,158	2,633,222,102
instructions:u	2350701799	2,416,421,797	31,926,323,047	31,690,702,949	5,968,631,635
cache-references:u	704417599	618,146,679	2,831,838,694	2,808,801,545	1,323,500,619
cache-misses:u	3585380	5,383,390	14,550,181	13,002,187	3,624,454
L1-dcache-loads:u	703979214	618,194,305	2,831,577,948	2,809,117,566	1,322,680,557
L1-dcache-load-misses:u	3568457	5,430,485	14,573,916	13,065,608	3,729,197
LLC-loads:u	4712306	6,666,961	18,582,957	17,159,504	4,688,008
LLC-load-misses:u	757910	3,418,870	7,207,590	6,341,038	1,105,372

表 8: 不同架构下性能指标对比（次）

事件类型	1 线程	2 线程	4 线程	8 线程
cpu-clock:u	1,304,000,000	480,000,000	612,000,000	708,000,000
cycles:u	513,852,298	701,603,178	929,112,981	589,766,908
instructions:u	1,173,219,186	1,515,201,749	1,552,759,867	1,269,919,408
cache-references:u	360,945,996	468,694,243	467,450,601	407,174,978
cache-misses:u	402,049	1,222,909	2,002,497	851,506
L1-dcache-loads:u	370,089,880	468,898,236	462,969,160	413,938,930
L1-dcache-load-misses:u	395,490	1,865,515	2,035,306	840,330
LLC-loads:u	264,326	2,120,694	4,342,655	816,264
LLC-load-misses:u	109,036	373,683	950,000	954,328

表 9: 不同线程数下的事件总量（次）