南開大學
Nankai University

南开大学

计算机学院和密码与网络空间安全学院

《并行程序设计》实验报告

作业四：NTT 算法 Pthread&OpenMP 编程并行优化

姓名：梁景铭

学号：2312632

专业：计算机科学与技术

指导教师：王刚

2025 年 5 月 13 日

摘要

本报告重点探讨了基于 Pthread 和 OpenMP 两种并行编程模型对 NTT 算法进行优化的方法。

研究内容包括 NTT 算法本身的并行化方向，以及结合中国剩余定理 (CRT) 进行多线程优化的策略。在 Pthread 部分，探索了静态线程和动态线程的实现方式，并对比了它们在不同模数及数据规模下的性能表现，特别关注了 CRT 优化在其中的作用和局限性。在 OpenMP 部分，研究了 OpenMP 的并行模型、指令、数据属性等，并将其应用于 NTT 算法的并行实现，同时也探讨了 OpenMP 结合 CRT 的优化方案。

通过实验对比和性能分析，包括使用 Perf 工具进行 Profiling，我们评估了不同并行实现方案的性能优劣，并提出了一些个人的思考。实验代码及图片已全部上传至：

https://github.com/eprogressing/NKU_COSC0025_Parallel

关键词：NTT, 并行计算, Pthread, OpenMP, 多线程, CRT, 性能优化

目录

1 问题描述

1.1 期末选题

NTT，Number Theoretic Transform，数论变换。这种算法是以数论为基础，对样本点的数论变换，按时间抽取的方法，得到一组等价的迭代方程，有效高速地简化了方程中的计算公式。与直接计算相比，大大减少了运算次数。数论变换是一种计算卷积的快速算法。

NTT 具有高效性、适用性和可扩展性等特点，可以应用于信号处理、图像处理、密码学等领域。相比于传统的算法，NTT 能够大大减少计算运算次数，提高计算效率，是解决一些特定问题的有力工具。在期末大作业中，拟在之前实验基础上探索更多的 NTT 优化算法 [1]，对比不同并行环境下不同算法的性能，并在特定的场景下应用。

NTT 的计算流程图，如图 1.1 所示。

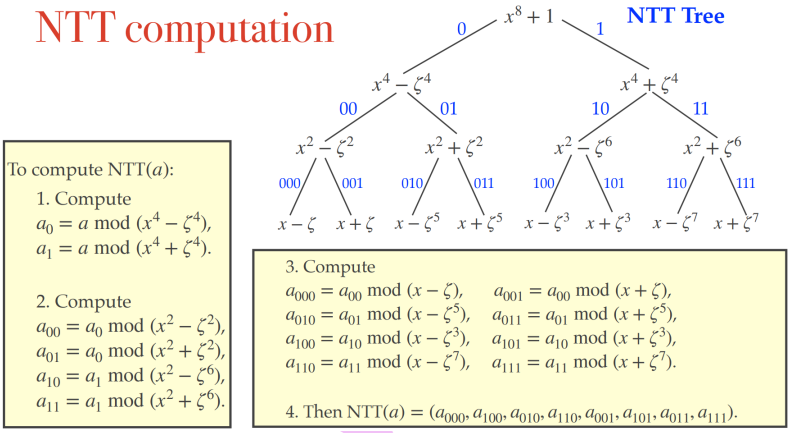


图 1.1: NTT 计算流程图 [2]

这张图直观展示了 NTT 的分治计算流程：通过递归模运算将复杂多项式拆解为线性因子，最终组合得到变换结果。右侧的树形结构体现了分治的层次性，左侧步骤则具体说明了每层分解的操作。

1.2 本次题目选题

1.2.1 实验要求

本次多线程 NTT 实验旨在对比三种不同算法在超大模数情景下的性能与可扩展性，并结合 OpenMP 实现多线程加速，在报告中对各方案性能差异与多线程加速效果进行分析。实验要求如下：

- 实现三种 NTT 算法方案：朴素算法、四分算法、CRT 算法
  - 必须实现朴素算法，若实现 CRT 算法可无需实现四分算法
  - 最终确保 CRT 算法得分 ≥ 四分算法得分
- 多线程优化：将原 Pthread 实验改写为 OpenMP，编译需加 `-fopenmp` 选项
- CRT 多模数合并思路：为各线程分配不同小模数，计算后通过 CRT 合并，适用于超大模数（> 32 位，实验必须实现）
- 实验中禁止使用 SIMD 优化，向量化内容可在期末报告中展开

2 NTT 算法

2.1 FTT、NTT 算法对比分析

在算法导论课中，我们已经学习过 FFT 算法，这里只做简单的描述。快速傅里叶变换（FFT）是离散傅里叶变换（DFT）的高效算法，可将复杂度从 $O(N^2)$ 降至 $O(N \log N)$。

- 分治法：将 DFT 分解为更小的 DFT
- 利用旋转因子 $W_N^{nk} = e^{-j2\pi nk/N}$ 的对称性和周期性
- 常用 Cooley-Tukey 算法

2.1.1 关键公式

DFT 定义：

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{nk}, \quad 0 \le k \le N-1$$

FFT 通过分解为：

$$\begin{cases} X[2r] = \sum_{m=0}^{N/2-1} (x[m] + x[m+N/2]) \cdot W_{N/2}^{mr} \\ X[2r+1] = \sum_{m=0}^{N/2-1} (x[m] - x[m+N/2]) \cdot W_N^m \cdot W_{N/2}^{mr} \end{cases}$$

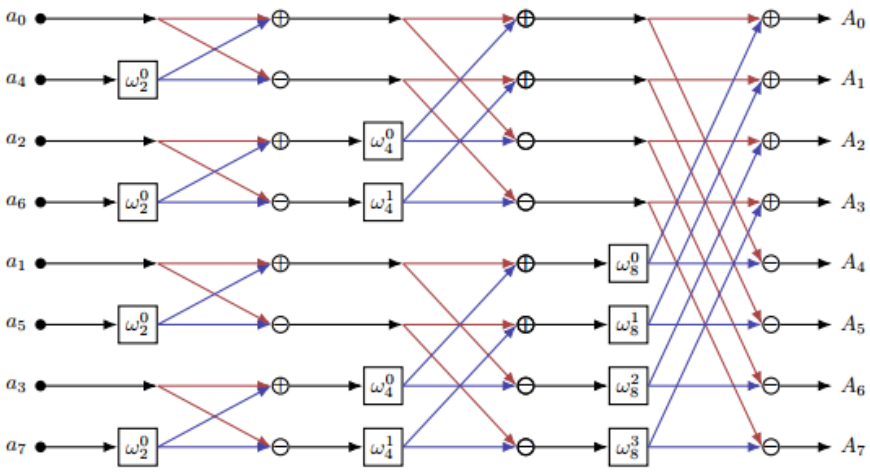


图 2.2: FFT 算法解释图

而 NTT 是 FFT 在数论基础上的实现。尽管 FFT 在多项式乘法等计算问题中具有 $O(n \log n)$ 的优良时间复杂度，但其仍存在一些不足之处。

首先，FFT 依赖于复数域的离散傅里叶变换，其计算过程中不可避免地使用了浮点数。在实际计算中容易造成舍入误差的积累，尤其在进行正变换和逆变换之后，误差可能导致结果偏离整数，影响精度。在一些对数值结果要求严格的场景，如大整数乘法、组合计数或模意义下的系数计算中，这种精度问题尤为突出。

此外，FFT 的实现涉及大量复数运算，包括旋转因子的计算、实部与虚部的处理等，增加了实现的复杂性，并导致程序运行中的常数较大。这在对运行效率要求极高的场景中可能成为瓶颈。

为了解决上述问题，引入了数论变换（NTT）。NTT 与 FFT 在理论结构上非常相似，但其运算完全基于模 p 的整数域，使用整数单位根替代复数单位根，从而彻底避免了浮点数带来的精度误差。同时，由于所有运算均为整数加法、减法与乘法，NTT 在实现上更加简洁，且具有更小的常数开销。在许多面向整数计算的问题中，NTT 不仅保证了结果的准确性，也提升了整体的计算效率。

2.1.2 FFT，NTT 计算开销对比

这里我通过 Leetcode 第 43 题字符串相乘运行程序，对比 FFT 和 NTT 之间的开销，结果如下：

图 2.3: FFT 和 NTT 程序性能对比

可以看到在执行用时分布和内存消耗分布上，NTT 都比 FTT 有更加明显的优势。

2.2 NTT 算法分析

2.2.1 数论前置知识

欧拉函数和欧拉定理　欧拉函数 $\varphi(n)$ 表示小于等于 n 且与 n 互素的正整数的个数。特别地，当 n 是素数时，有：

$$\varphi(n) = n - 1.$$

欧拉定理指出：对于任意整数 $a \in \mathbb{Z}$ 和正整数 $m \in \mathbb{N}^*$，若 $\gcd(a, m) = 1$，则有：

$$a^{\varphi(m)} \equiv 1 \pmod{m}.$$

在 NTT 中，通常需要在模一个质数 p 的意义下进行运算。此时，$\varphi(p) = p - 1$，欧拉定理保证了对于任意与 p 互素的 a，有 $a^{p-1} \equiv 1 \pmod{p}$。这一性质使得我们可以选取一个原根 g，通过 $g^{(p-1)/n}$ 构造出 n 次单位根，从而实现模 p 的快速数论变换。

费马小定理　费马小定理指出：若 p 为素数，且 $\gcd(a, p) = 1$，则有

$$a^{p-1} \equiv 1 \pmod{p}.$$

等价地，对于任意整数 a，有

$$a^p \equiv a \pmod{p}.$$

在数论变换中，为了进行模 p 意义下的除法操作，常常需要求某个数的逆元。由费马小定理可得，当 a 与 p 互素时，其模逆元为

$$a^{-1} \equiv a^{p-2} \pmod{p}.$$

这使得我们可以通过快速幂算法高效地求出逆元，从而完成 NTT 中对多项式系数的归一化处理。

阶　由欧拉定理可知，若 $\gcd(a,m)=1$，则存在最小的正整数 n 使得

$$a^n \equiv 1 \pmod{m}.$$

这个最小的 n 称为 a 模 m 的**阶**，记作 $\delta_m(a)$ 或 $ord_m(a)$。

阶具有以下两个重要性质：

- **性质 1：** $a, a^2, \ldots, a^{\delta_m(a)}$ 在模 m 意义下两两不同余，之后将进入周期性重复；
- **性质 2：** 若 $a^n \equiv 1 \pmod{m}$，则 $\delta_m(a) \mid n$，从而可推得若 $a^p \equiv a^q \pmod{m}$，则 $p \equiv q \pmod{\delta_m(a)}$。

在 NTT 中，为了构造 n 次单位根 ω，需选择一个模 p 的原根 g，使得 $ord_p(g) = p-1$，再令 $\omega = g^{(p-1)/n}$，确保 ω 的阶为 n，从而能够生成 n 个不同的单位根，满足数论变换的需求。

原根　设 $n \in \mathbb{N}^*$，$g \in \mathbb{Z}$，若 $\gcd(g,m)=1$ 且 $\delta_m(g) = \varphi(m)$，则称 g 为模 m 的**原根**。

当 m 为素数时，有 $\varphi(m) = m-1$，此时 $g^i \bmod m$ 对于 $0 < i < m$ 两两不同。

原根个数： 若模数 m 存在原根，则原根的个数为 $\varphi(\varphi(m))$。

原根存在定理： 模数 m 存在原根当且仅当 $m = 2, 4, p^a, 2p^a$，其中 p 为奇素数，$a \in \mathbb{N}^*$。

原根的性质：

- **不重性：** $\forall\ 0 \le i < j < \varphi(p)$，有 $g^i \not\equiv g^j \pmod{p}$；
- **折半性：** 定义 $g_n = g^{\frac{p-1}{n}}$，则有 $g_{an}^{ak} \equiv g_n^k \pmod{p}$；
- **对称性：** $g_{2n}^{k+n} \equiv -g_{2n}^k \pmod{p}$；
- **求和性：** $\sum_{i=0}^{n-1} (g_n)^{ki} \equiv n[k=0] \pmod{p}$，其中 $[k=0]=1$，否则为 0。

原根与模数的选择： 为了支持多次二分变换，模数 p 一般选取为形如 $p = q \cdot 2^k + 1$ 的素数，其中 q 为奇素数，k 控制可支持的最大变换长度 2^k。

表 1：常用模数与原根

原根 g	模数 p	分解形式	模数的阶
3	469762049	$7 \times 2^{26} + 1$	2^{26}
3	998244353	$119 \times 2^{23} + 1$	2^{23}
3	2281701377	$17 \times 2^{27} + 1$	2^{27}

2.3　NTT 串行算法实现

现在给出迭代版本的算法，不难发现除了将单位根替换为原根，增加模运算，以及增加了参数，原根 g，模数 p 外与 FFT 并无太大区别。

Algorithm 1 Iteration NTT

Input: Array $A = [a_0, a_1, \ldots, a_{n-1}]$, length $n = 2^k$, primitive root g, prime modulus p
Output: NTT transformed array P

```
n ← len(A)  P ← BitReverseCopy(A)  for s = 1 to log n do
    m ← 2^s  g_m ← g^{n/m} mod p  for k = 0 to n − 1 m do
        φ ← 1  for j = 0 to m/2 − 1 do
            t ← φ · P[k + j + m/2] mod p  u ← P[k + j] mod p  P[k + j] ← (u + t) mod p  P[k + j + m/2] ←
              (u − t) mod p  φ ← (φ · g_m) mod p
return P
```

自此，关于 NTT 算法的介绍和分析就结束了，下面的部分将根据实验要求和上面的叙述，进行代码的优化和 Pthread& OpenMP 编程并行操作。

3 Pthread 编程范式

Pthread（POSIX Threads）是 POSIX 标准定义的线程接口，提供了创建、管理和同步线程的底层 API，较 OpenMP 更灵活但也更复杂。

3.1 头文件与编译选项

- 头文件：`#include <pthread.h>`
- 链接选项：`-lpthread` 或`-pthread`

3.2 主要 API

Listing 1: Pthread 核心函数原型

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   void *arg);
int pthread_join(pthread_t thread,
                 void **retval);
void pthread_exit(void *retval);
int pthread_cancel(pthread_t thread);
```

Pthread 编程常见两大范式，分别是 **静态线程**与 **动态线程**。

3.3 静态线程范式

- 在程序初始化阶段创建固定数量的工作线程。
- 通过任务队列分发子任务给空闲线程。
- 主线程只负责初始化和最终回收，不在每个并行段创建 / 销毁线程。

Listing 2: 静态线程池示例框架

基本框架

```
pthread_mutex_t queue_mutex;
pthread_cond_t queue_cond;
int            task_queue[MAX_TASKS];
int            head = 0, tail = 0;
void *worker(void *arg) {
    for (;;) {
        pthread_mutex_lock(&queue_mutex);
        while (head==tail)
```

```
            pthread_cond_wait(&queue_cond, &queue_mutex);
        int task = task_queue[head++];
        pthread_mutex_unlock(&queue_mutex);
        if (task < 0) break;
        process_task(task);
    }
    return NULL;
}
int main() {
    for (int i = 0; i < N_THREADS; i++)
        pthread_create(&threads[i], NULL, worker, NULL);
    for each parallel_section {
        for (每个子任务 i) {
            pthread_mutex_lock(&queue_mutex);
            task_queue[tail++] = i;
            pthread_mutex_unlock(&queue_mutex);
            pthread_cond_signal(&queue_cond);
        }
    }
    for (int i = 0; i < N_THREADS; i++) {
        pthread_mutex_lock(&queue_mutex);
        task_queue[tail++] = -1;
        pthread_mutex_unlock(&queue_mutex);
        pthread_cond_signal(&queue_cond);
    }
    for (i=0; i<N_THREADS; i++)
        pthread_join(threads[i], NULL);
    return 0;
}
```

3.4 动态线程范式

- 在每个并行段开始时，主线程调用 pthread_create 创建线程组。
- 子任务执行完毕后，通过 pthread_join 回收线程。
- 适用于并行段少且开销可接受的场景。

Listing 3: 动态线程示例框架

基本框架

```
void *worker(void *arg) { /* ... */ }
int main() {
    for each parallel_section {
        for (i=0; i<NT; i++)
```

```
            pthread_create(&threads[i], NULL, worker, &args[i]);
        for (i=0; i<NT; i++)
            pthread_join(threads[i], NULL);
    }
    return 0;
}
```

4 NTT Pthread 多线程优化

4.1 Pthread 程序优化方向

为获得高性能并行，对 Pthread 程序可从以下方面优化：

1. **最小化线程创建 / 销毁开销**:
   - 优先采用静态线程池，避免在高频并行段中重复创建销毁。
   - 对于必须动态创建的场景，可考虑自定义线程缓存。
2. **减少同步开销**:
   - 使用尽量粗粒度或无锁算法，减少 `pthread_mutex_lock/unlock` 次数。
   - 合理使用读写锁 (`pthread_rwlock`) 或原子操作。
   - 缩小临界区范围，尽量将同步原语围绕必要代码。
3. **负载均衡与任务划分**:
   - 根据任务执行时间，动态分配工作。
   - 减少” 最后一个线程” 的空闲等待现象。
4. **减少内存伪共享**:
   - 为线程私有数据或频繁写入的计数器使用对齐填充。
   - 避免多个线程频繁写入同一缓存行。
5. **线程亲和性**:
   - 通过 `pthread_attr_setaffinity_np` 将线程绑定到特定核，减少缓存抖动。
6. **合理使用条件变量和屏障**:
   - 对于需要多阶段同步的场景，使用 `pthread_barrier_t`。
   - 避免不必要的广播，优先使用 `pthread_cond_signal`。

4.2 NTT 朴素多线程优化

为提高多项式乘法中 NTT 的执行效率，我们利用 pthread 将正向和逆向 NTT 分别并行化处理，具体优化策略如下：

1. **划分变换任务**:
   - 将输入多项式的正变换任务分成两部分：分别对数组 `A` 和 `B` 并行执行正向 NTT。

- 逆变换时，由于需要对点积结果进行一次 NTT，故使用单线程执行。

2. **线程创建与同步**：
   - 定义线程参数结构体 `NTTThreadArg`，包含 `a`（数据指针）、`n`（变换长度）、`P`（模数）、`G`（原根）、`type`（1 表示正变换，-1 表示逆变换）。
   - 使用 `pthread_create` 启动两个线程，分别调用 `ntt_thread` 完成对 `A`、`B` 的正向 NTT。
   - 主线程通过 `pthread_join` 等待两个子线程结束，确保两者都完成后再进行点乘操作。
   - 点乘完成后，再次调用 `pthread_create` 启动一个线程执行逆向 NTT，并使用 `pthread_join` 等待其完成。

3. **数据组织与内存布局**：
   - 为保证线程安全与内存访问局部性，正逆变换均在独立的、预先分配好的 `vector<int>` 缓冲区中完成，避免多线程下的数据竞态。
   - 变换结果直接写回原缓冲区，点乘结果原地保存，减少额外拷贝开销。

可以用一句话概括为：将正向 NTT 划分为两个并行线程执行，待完成后汇总点乘，再以单线程执行逆向 NTT，从而通过 pthread 提升多项式乘法的并行度和性能。代码如下：

```cpp
#include <cstdio>
#include <cstring>
#include <cmath>
#include <algorithm>
#include <chrono>
#include <thread>
#include <vector>
#include <fstream>
#include <iostream>
#include <pthread.h>
using namespace std;
using int64 = long long;
int64 modpow(int64 a, int64 e, int64 m) {
    int64 r = 1;
    a %= m;
    while (e) {
        if (e & 1) r = (__int128)r * a % m;
        a = (__int128)a * a % m;
        e >>= 1;
    }
    return r;
}
// 找 P 的一个原根
int get_primitive_root(int P) {
    int phi = P - 1, n = phi;
    vector<int> fac;
```

```
    for (int i = 2; i * i <= n; ++i) {
        if (n % i == 0) {
            fac.push_back(i);
            while (n % i == 0) n /= i;
        }
    }
    if (n > 1) fac.push_back(n);
    for (int g = 2; g < P; ++g) {
        bool ok = true;
        for (int f : fac) {
            if (modpow(g, phi / f, P) == 1) {
                ok = false; break;
            }
        }
        if (ok) return g;
    }
    return -1;
}
// 单次 NTT (type=1 正变换, type=-1 反变换)
void ntt(int *a, int n, int P, int G, int type) {
    int L = __builtin_ctz(n);
    vector<int> rev(n);
    rev[0] = 0;
    for (int i = 1; i < n; ++i)
        rev[i] = (rev[i>>1]>>1) | ((i&1)<<(L-1));
    for (int i = 0; i < n; ++i)
        if (i < rev[i]) swap(a[i], a[rev[i]]);

    for (int len = 1; len < n; len <<= 1) {
        int64 wn = modpow(G, (P - 1) / (len << 1), P);
        if (type == -1) wn = modpow(wn, P - 2, P);
        for (int i = 0; i < n; i += (len << 1)) {
            int64 w = 1;
            for (int j = 0; j < len; ++j) {
                int64 u = a[i+j];
                int64 v = (__int128)w * a[i+j+len] % P;
                a[i+j]     = (u + v) % P;
                a[i+j+len] = (u - v + P) % P;
                w = w * wn % P;
            }
        }
    }
```

```
    if (type == -1) {
        int64 inv_n = modpow(n, P - 2, P);
        for (int i = 0; i < n; ++i)
            a[i] = (__int128)a[i] * inv_n % P;
    }
}
struct NTTThreadArg {
    int *a;
    int n;
    int P, G;
    int type; // 1 or -1
};
void* ntt_thread(void* _arg) {
    auto arg = (NTTThreadArg*)_arg;
    ntt(arg->a, arg->n, arg->P, arg->G, arg->type);
    return nullptr;
}
// 基于 pthread 的多线程 NTT 多项式乘法
void poly_multiply(int *a, int *b, int *out, int n, int P) {
    int len = 1;
    while (len < 2 * n) len <<= 1;
    vector<int> A(len, 0), B(len, 0);
    for (int i = 0; i < n; ++i) {
        A[i] = (a[i] % P + P) % P;
        B[i] = (b[i] % P + P) % P;
    }
    int G = get_primitive_root(P);
    pthread_t t1, t2;
    NTTThreadArg arg1{A.data(), len, P, G, 1};
    NTTThreadArg arg2{B.data(), len, P, G, 1};
    pthread_create(&t1, nullptr, ntt_thread, &arg1);
    pthread_create(&t2, nullptr, ntt_thread, &arg2);
    pthread_join(t1, nullptr);
    pthread_join(t2, nullptr);
    for (int i = 0; i < len; ++i)
        A[i] = (int)((int64)A[i] * B[i] % P);
    NTTThreadArg arg3{A.data(), len, P, G, -1};
    pthread_create(&t1, nullptr, ntt_thread, &arg3);
    pthread_join(t1, nullptr);
    for (int i = 0; i < 2 * n - 1; ++i)
        out[i] = A[i];
}
```

4.3 CRT 多线程优化

4.3.1 CRT 合并大模数原理

假设需要对大模数 $P = M_1 M_2 \cdots M_k$ 下的多项式做 NTT，其中各 M_i 两两互素，且都适合快速 NTT。记在每个小模数下的变换结果为

$$C^{(i)}(x) \equiv A(x) \cdot B(x) \bmod (x^n - 1,\, M_i), \quad i = 1, 2, \ldots, k.$$

则可以通过中国剩余定理将各分量合并为大模数下的结果：

$$C(x) \equiv \sum_{i=1}^{k} C^{(i)}(x)\, T_i \mod P,$$

其中

$$T_i \equiv \Big(\prod_{j \neq i} M_j\Big) \times \Big(\prod_{j \neq i} M_j\Big)^{-1} \bmod M_i$$

为重建系数，满足

$$T_i \equiv \begin{cases} 1 \pmod{M_i}, \\ 0 \pmod{M_j} \quad (j \neq i). \end{cases}$$

因此，完整的 CRT 合并公式为

$$C(x) \;=\; \left(\sum_{i=1}^{k} C^{(i)}(x) \left(\tfrac{P}{M_i}\right) \left(\tfrac{P}{M_i}\right)^{-1} \bmod M_i \right) \bmod P.$$

在下面的实验中，我们采用三模数 NTT 合并。

4.3.2 动态线程

为了对超大模数 P 下的多项式乘法进行高效计算，本方案引入三模数分解与 CRT 合并，结合 pthread 并行化，主要流程如下：

1. **模数与原根选择**

   选取三个互素且均形如 $c \cdot 2^k + 1$ 的小模数：

   $$\texttt{mod}_1 = 7\,340\,033, \quad \texttt{mod}_2 = 104\,857\,601, \quad \texttt{mod}_3 = 469\,762\,049,$$

   均使用公共原根 $G = 3$。

2. **单模 NTT 变换函数**

   - 对长度为 n（为二的幂）的输入数组 $\{a_i\}$ 实现 Cooley–Tukey 迭代 NTT：

     $$\texttt{ntt}(a, n, \text{mod}, \text{type})$$

   - type $= 1$ 时为正变换，type $= -1$ 时为逆变换，逆变换最后乘以 $n^{-1} \pmod{\text{mod}}$。

3. **三线程并行执行单模 NTT**

- 定义线程参数结构体 `NTTArgs`$\{a, b, result, n, mod\}$，每个线程分别对 $\texttt{mod}_1, \texttt{mod}_2, \texttt{mod}_3$ 执行：

$$\texttt{pthread\_create}(\ldots, \texttt{ntt\_thread}, \&\texttt{args}[i]),$$

- 线程内部：对输入多项式零扩展到长度 len 后，执行

$$\texttt{ntt}(A, len, \text{mod}_i, 1), \quad \texttt{ntt}(B, len, \text{mod}_i, 1), \quad A_j \leftarrow A_j \cdot B_j, \quad \texttt{ntt}(A, len, \text{mod}_i, -1);$$

  并将结果写入 `result` 缓冲区。
- 主线程等待所有三路变换完成：

$$\texttt{pthread\_join}(\ldots) \times 3.$$

4. **两步 CRT 合并**

(a) 第一步：合并 $\texttt{mod}_1$ 与 $\texttt{mod}_2$

$$t \equiv (a_2 - a_1)\, m_1^{-1} \pmod{m_2}, \quad x_{12} = a_1 + t \cdot m_1 \pmod{m_1 m_2}.$$

(b) 第二步：将 x_{12} 与 $\texttt{mod}_3$ 合并

$$t' \equiv (a_3 - x_{12})\,(m_1 m_2)^{-1} \pmod{m_3}, \quad x \equiv x_{12} + t' \cdot (m_1 m_2) \pmod{P}.$$

(c) 其中逆元预先计算：

$$m_1^{-1} \bmod m_2, \quad (m_1 m_2)^{-1} \bmod m_3$$

利用扩展欧几里得法在编译期初始化。

5. **整体多项式乘法接口**

$$\texttt{poly\_multiply}(a, b, \texttt{out}, n, P)$$

(a) 确定变换长度 $len \geq 2n$ 的最小二幂；

(b) 零扩展输入，多线程并行单模 NTT 乘积；

(c) CRT 合并三路结果至 $\texttt{out}[0 \ldots 2n-2]$。

该方案利用三路并行 NTT 分担不同小模运算，再通过 CRT 恢复大模结果，线程间无数据竞争，适合超大模数的高效乘法。核心代码如下：

```
static const int64 mod1 = 7340033, mod2 = 104857601, mod3 = 469762049, G = 3;
// 预计算 CRT 逆元
static const int64 inv_m1_mod_m2 = [](){
    int64 a=mod1,b=mod2,u=1,v=0;
    while(b){ int64 t=a/b; a-=t*b; swap(a,b); u-=t*v; swap(u,v); }
    return (u%mod2+mod2)%mod2;
}();
```

```
static const int64 inv_m12_mod_m3 = [](){
    int64 m12 = mod1*mod2%mod3, a=m12,b=mod3,u=1,v=0;
    while(b){ int64 t=a/b; a-=t*b; swap(a,b); u-=t*v; swap(u,v); }
    return (u%mod3+mod3)%mod3;
}();
struct NTTArgs { int *a, *b, *result; int n, mod; };
void* ntt_thread(void* _args) {
    auto args = (NTTArgs*)_args;
    vector<int> A(args->a, args->a+args->n), B(args->b, args->b+args->n);
    ntt(A.data(), args->n, args->mod, 1);
    ntt(B.data(), args->n, args->mod, 1);
    for(int i=0;i<args->n;++i) A[i]=int((int64)A[i]*B[i]%args->mod);
    ntt(A.data(), args->n, args->mod, -1);
    memcpy(args->result, A.data(), args->n*sizeof(int));
    return nullptr;
}
// 两步 CRT 合并
void crt_merge(int* r1,int* r2,int* r3,int* out,int n,int P){
    int64 m1=mod1,m2=mod2,m3=mod3,m12=m1*m2;
    for(int i=0;i<2*n-1;++i){
        int64 a1=r1[i], a2=r2[i], a3=r3[i];
        int64 t = ((a2-a1)%m2+m2)%m2 * inv_m1_mod_m2 % m2;
        __int128 x12 = a1 + (__int128)t*m1;
        int64 t2 = ((a3- (int64)(x12%m3))%m3+m3)%m3 * inv_m12_mod_m3 % m3;
        __int128 x = x12 + (__int128)t2*m12;
        out[i] = (int)(x % P);
    }
}
// 多线程 NTT + CRT 的多项式乘法接口
void poly_multiply(int *a, int *b, int *out, int n, int P) {
    int len=1; while(len<2*n) len<<=1;
    vector<int> a_pad(len), b_pad(len), r1(len), r2(len), r3(len);
    copy(a,a+n,a_pad.begin()); copy(b,b+n,b_pad.begin());
    NTTArgs args[3] = {
      {a_pad.data(), b_pad.data(), r1.data(), len, (int)mod1},
      {a_pad.data(), b_pad.data(), r2.data(), len, (int)mod2},
      {a_pad.data(), b_pad.data(), r3.data(), len, (int)mod3}
    };
    pthread_t th[3];
    for(int i=0;i<3;++i) pthread_create(&th[i], nullptr, ntt_thread, &args[i]);
    for(int i=0;i<3;++i) pthread_join(th[i], nullptr);
    crt_merge(r1.data(), r2.data(), r3.data(), out, n, P);
```

```
}
```

4.3.3 静态线程

静态线程范式在程序启动阶段即创建固定数量的工作线程，并在整个运行期中保持这些线程存活。当有并行任务到来时，将任务分派到空闲线程执行；任务完成后，线程不退出，而是返回线程池等待下一次分派。该方式优点是避免了频繁的线程创建与销毁开销，提高了整体性能和响应速度；缺点是线程池始终占用系统资源，若任务量波动大或并行区域稀疏，会导致部分线程长时间空闲，浪费资源。

静态线程在程序启动时创建，并在整个程序生命周期内保持活动，以避免频繁创建和销毁线程的开销，特别适用于需要重复执行相同任务的场景。在程序启动时，使用 `pthread_create` 函数创建三个静态线程 `t1`、`t2`、`t3`，分别调用 `worker1`、`worker2`、`worker3` 函数，每个线程负责处理一个特定的模数下的 NTT 计算，代码如下：

```
pthread_t t1, t2, t3;
pthread_create(&t1, nullptr, worker1, nullptr);
pthread_create(&t2, nullptr, worker2, nullptr);
pthread_create(&t3, nullptr, worker3, nullptr);
```

每个线程执行一个无限循环，等待对应的 `ready` 标志（`ready1`、`ready2`、`ready3`）被设置为真。一旦标志为真，线程会复制全局输入数据 `g_a_pad` 和 `g_b_pad`，执行正向 NTT，进行点乘，执行逆 NTT，将结果存储到 `g_res1`、`g_res2` 或 `g_res3`，设置对应的 `done` 标志（`done1`、`done2`、`done3`），然后重置 `ready` 标志。以 `worker1` 为例，其实现如下：

```
void* worker1(void*) {
    while (true) {
        if (ready1) {
            vector<int64_t> a = g_a_pad;
            vector<int64_t> b = g_b_pad;
            ntt(a, g_len, mod1, false);
            ntt(b, g_len, mod1, false);
            for (int i = 0; i < g_len; i++) {
                a[i] = a[i] * b[i] % mod1;
            }
            ntt(a, g_len, mod1, true);
            g_res1 = a;
            done1 = true;
            ready1 = false;
        }
    }
    return nullptr;
}
```

同步机制使用原子布尔变量 `ready1`、`ready2`、`ready3` 指示线程开始任务，`done1`、`done2`、`done3`

指示任务完成，主线程通过轮询 done 标志等待所有线程完成。全局变量 g_a_pad、g_b_pad 用于存储填充后的输入多项式，g_len 表示变换长度，g_res1、g_res2、g_res3 保存线程计算结果。主线程负责准备输入数据并填充到 g_a_pad 和 g_b_pad，设置 ready 标志启动线程，轮询 done 标志直到所有线程完成，然后使用 CRT 合并结果。这种设计减少了线程创建和销毁的开销，三个线程并行处理不同模数下的 NTT 加速了计算，使用原子标志进行轻量级同步避免了锁的开销，从而在多核系统上实现了高效的并行 NTT 计算，适用于高性能计算场景。

4.4 不同模数下 CRT 运行结果对比

在三模数系统中，程序需要创建并管理三个静态线程来并行处理不同模数下的 NTT 计算，这涉及到线程的创建、销毁和同步开销，而二模数系统只需管理两个线程，一模数系统则完全在主线程中完成计算，无需额外线程管理。

随着模数数量减少，线程创建与销毁的系统开销逐渐降低，同时同步开销也因同步点减少而缩短。在三模数系统中，主线程通过轮询 done 标志等待三个线程完成任务，这种机制会占用 CPU 资源，尤其在任务执行时间较长时开销更大，而二模数系统只需等待两个线程，一模数系统则完全无需同步。

另一方面，CRT 合并是恢复最终结果的关键步骤，其计算量与模数数量直接相关：三模数系统需要两次合并操作，涉及多次模逆和乘法运算；二模数系统只需一次合并；一模数系统则无需合并，直接使用单个模数结果。

因此，随着模数数量从三模数减少到二模数再到一模数，线程管理、同步和 CRT 合并的开销逐步减少，运行速度显著提升，尤其在一模数系统中性能最佳。这一发现为优化 NTT 并行计算提供了重要参考，特别是在资源受限或性能要求高的场景中。结果如下：

表 1: 不同模数配置下的运行结果

模数类型	n	p	平均延迟 (μs)
单模数	4	7340033	0.00947
单模数	131072	7340033	119.966
单模数	131072	104857601	119.493
单模数	131072	469762049	119.98
二模数	4	7340033	0.25974
二模数	131072	7340033	134.157
二模数	131072	104857601	133.476
二模数	131072	469762049	133.702
三模数	4	7340033	0.260551
三模数	131072	7340033	196.931
三模数	131072	104857601	196.147
三模数	131072	469762049	195.546

4.5 大于 32 bit 的模数测试

新代码采用单模数 1337006139375617，取代了之前的小于 32bit 单模数配置。以下是两者的主要区别：

新代码使用了一个大于 64 位的超大模数 1337006139375617，其分解为 $2^5 \times 3^2 \times 464074081$，确保了其为一个大质数，并与原根 $G = 3$ 兼容。而之前的代码使用多个较小的 32 位模数（例如 7340033、104857601、469762049），通过中国剩余定理（CRT）合并结果。

新代码简化了多模数并行架构，仅在单模数下执行 NTT。线程使用上，新代码创建了两个线程分别处理输入多项式的正向 NTT。此外，新代码在 ‘poly_multiply‘ 函数中加入了长度限制检查（‘if (len > 32) throw runtime_error(...)‘），表明对 NTT 适用性的限制，而之前的代码未明确限制长度。计算过程中，新代码的逆 NTT 是串行执行的。数据类型上，新代码使用 ‘int64‘（64 位整数）并依赖 ‘__int128‘ 进行大数运算。具体改进代码如下：

4.5.1 模数定义

```
static const int64 mod = 1337006139375617;
static const int64 G = 3;
```

4.5.2 线程使用

```
struct NTTThreadArg {
    int64 *data;
    int n;
    int64 mod;
};
void* ntt_forward_thread(void* _arg) {
    auto arg = (NTTThreadArg*)_arg;
    vector<int64> A(arg->data, arg->data + arg->n);
    ntt(A.data(), arg->n, arg->mod, 1);
    memcpy(arg->data, A.data(), arg->n * sizeof(int64));
    return nullptr;
}
NTTThreadArg args[2] = {
    {a_pad.data(), len, mod},
    {b_pad.data(), len, mod}
};
pthread_t threads[2];
for (int i = 0; i < 2; ++i) {
    pthread_create(&threads[i], nullptr, ntt_forward_thread, &args[i]);
}
for (int i = 0; i < 2; ++i) {
    pthread_join(threads[i], nullptr);
}
```

4.5.3 CRT 相关

```
int len = 1;
while (len < 2 * n) len <<= 1;
```

```
if (len > 32) {
    throw runtime_error("NTT not supported for len > 32 with this modulus");
}
for (int i = 0; i < 2 * n - 1; ++i)
    out[i] = a_pad[i] % P;
```

4.6 为什么使用 CRT 而不是朴素多线程

4.6.1 朴素多线程的局限性

朴素多线程方法试图通过将单一模数下的 NTT 计算分配给多个线程来加速。然而，这种方法受限于数据依赖性，需要频繁的线程同步，导致并行效率低下。此外，单一模数必须同时满足 NTT 友好的条件（即模数为素数且 $P-1$ 可被大幂次的 2 整除）和足够大的范围以避免系数溢出。例如，模数 $P = 1337006139375617$ 仅支持变换长度至 $2^5 = 32$，无法处理大 n。这些限制使得朴素多线程难以满足高性能和通用性需求。

4.6.2 CRT 的优势

CRT 通过在多个小模数（如 $m_1 = 7340033$、$m_2 = 104857601$、$m_3 = 469762049$）下并行执行 NTT，并利用 CRT 合并结果以得到目标模数 P 下的系数，克服了朴素多线程的局限性。其主要优势包括：

- **高效并行性**：每个模数的 NTT 计算完全独立，可分配给单独线程，无需同步。例如，三个模数可由三个线程并行处理 $O(n \log n)$ 的 NTT 计算，仅在最后的 CRT 合并阶段需要串行处理 $O(n)$ 操作，大幅提升并行效率。
- **模数灵活性**：CRT 允许使用多个 NTT 友好小模数，其乘积（如 $m_1 \cdot m_2 \cdot m_3 \approx 4.3 \times 10^{19}$）足以覆盖大模数 P 的平方，避免溢出。这支持任意 P 和更大的 n，无需单一模数同时满足严格约束。
- **计算效率**：小模数（23–29 位）支持快速的 32 位整数运算，相比大模数所需的 64 位或 $\_\_int128$ 运算，计算速度更快，缓存命中率更高。线程处理小模数任务也更高效。

5 NTT OpenMP 编程并行优化

5.1 OpenMP 编程范式

OpenMP 是一种针对共享内存架构的并行编程接口规范，通过在代码中插入编译指示（pragma）来实现并行化，无需修改底层算法。

5.1.1 并行模型

OpenMP 基于“主-从”模型：当遇到并行区域时，master 线程创建线程团队，并由所有线程并行执行该区域；执行完毕后，线程团队解散，回到串行执行。

5.1.2 并行指令

常用指令包括：

- `\#pragma omp parallel`：定义并行区域，所有线程执行。
- `\#pragma omp for`：将紧随其后的循环分段并行执行。
- `\#pragma omp sections`：将不同代码块分配给不同线程。
- `\#pragma omp single`：仅一个线程执行指定代码段。
- `\#pragma omp critical`：定义临界区，防止数据竞争。

5.1.3 数据属性与共享环境

在并行区域中，变量的属性主要有：

- **shared**：线程共享同一内存副本，适用于多线程读取或无需私有化的数据。
- **private**：每个线程拥有独立副本，常用于循环索引等。
- **firstprivate / lastprivate**：分别用于初始化和结果回写。
- **reduction**：并行归约操作（如求和、最大值）。

5.1.4 同步与屏障

- `#pragma omp barrier`：显式屏障，线程汇合点。
- `#pragma omp atomic`：对单一更新操作进行原子处理，开销小于 critical。
- 在 `for`、`sections` 等指令后添加 `nowait` 可避免隐式屏障。

5.1.5 任务并行（Tasks）

OpenMP 3.0 引入任务模型：

```
#pragma omp task
do_work(data);
#pragma omp taskwait
```

自 OpenMP 4.0 起，支持 `depend` 关键字以指定任务依赖关系。

5.1.6 运行时环境与调度策略

- 环境变量：`OMP_NUM_THREADS`（线程数）、`OMP_SCHEDULE`（static / dynamic / guided）。
- API 函数：`omp_set_num_threads()`（设置线程数）、`omp_get_wtime()`（计时）等。

5.2 OpenMP 多线程优化

5.2.1 算法总体结构

本实现分为以下几步：

1. 对输入多项式向量进行补零至合适长度（len 为二次幂）。
2. 三路并行执行 NTT 变换 / 点乘 / 逆 NTT，分别在模 m_1, m_2, m_3 下完成，得到中间结果 $\{r_1, r_2, r_3\}$。
3. 并行 CRT 合并三个结果，恢复到最终模 P 下的多项式乘积。
4. 使用 OpenMP 的并行区域、并行循环和并行区块来加速上述各步骤。

5.2.2 并行 NTT 函数实现

```
void ntt(int *a, int n, int mod, int type) {
    int L = __builtin_ctz(n);
    vector<int> rev(n);
    for(int i=1;i<n;i++) rev[i] = (rev[i>>1]>>1) | ((i&1)<<(L-1));
    for(int i=0;i<n;i++)
        if(i<rev[i]) swap(a[i], a[rev[i]]);
    for(int len=1; len<n; len<<=1) {
        int64_t wn = modpow(G, (mod-1)/(len<<1), mod);
        if(type==-1) wn = modpow(wn, mod-2, mod);
        #pragma omp parallel for schedule(static)
        for(int i=0; i<n; i += len<<1) {
            int64_t w = 1;
            for(int j=0; j<len; j++) {
                int64_t u = a[i+j];
                int64_t v = w * a[i+j+len] % mod;
                a[i+j]     = (u + v) % mod;
                a[i+j+len] = (u - v + mod) % mod;
                w = w * wn % mod;
            }
        }
    }
    if(type==-1) {
        int64_t inv_n = modpow(n, mod-2, mod);
        #pragma omp parallel for schedule(static)
        for(int i=0;i<n;i++)
            a[i] = (int64_t)a[i] * inv_n % mod;
    }
}
```

- 使用 `#pragma omp parallel for schedule(static)` 对最内层的蝶形段并行。
- 位反转置换和归一化步同样也采用并行循环以加速数据预处理与后处理。

5.2.3 三模并行与 CRT 合并

```
void poly_multiply(int *a, int *b, int *out, int n, int P) {
    int len=1; while(len<2*n) len<<=1;
    vector<int> padA(len), padB(len), r1, r2, r3;
    #pragma omp parallel sections
    {
      #pragma omp section
```

```
        { /* mod1 下的 NTT/A · B/逆 NTT, 结果存 r1 */ }
        #pragma omp section
        { /* mod2 下的 NTT/B · B/逆 NTT, 结果存 r2 */ }
        #pragma omp section
        { /* mod3 下的 NTT/B · B/逆 NTT, 结果存 r3 */ }
    }
    crt_merge(r1, r2, r3, out, n, P);
}
```

```
void crt_merge(const vector<int>& r1, const vector<int>& r2,
               const vector<int>& r3, int* out, int n, int P) {
    #pragma omp parallel for schedule(static)
    for(int i=0;i<2*n-1;i++){
        int64_t t = ((r2[i]-r1[i])%mod2+mod2)%mod2;
        t = t * inv_m1_mod_m2 % mod2;
        __int128 x12 = r1[i] + t * (int64_t)mod1;
        int64_t t2 = ((r3[i] - (int64_t)(x12%mod3))%mod3+mod3)%mod3;
        t2 = t2 * inv_m12_mod_m3 % mod3;
        __int128 x = x12 + t2 * (__int128)mod1 * mod2;
        out[i] = (int)(x % P);
    }
}
```

- 使用 parallel for 将 CRT 合并的每个系数恢复并行化。
- 预先计算好逆元 inv_m1_mod_m2、inv_m12_mod_m3，减少循环内开销。

6 实验和结果分析

6.1 运行结果

6.1.1 静态线程和动态线程对比

下面是我本次实验的运行结果，第一张为三模数 CRT 动态线程运行结果。

```
[s2312632@master_ubss1 ntt]$  g++ main.cc -o main -O2 -fopenmp -lpthread -std=c++11 && ./main
多项式乘法结果正确
average latency for n = 4 p = 7340033 : 0.260551 (us)
多项式乘法结果正确
average latency for n = 131072 p = 7340033 : 196.931 (us)
多项式乘法结果正确
average latency for n = 131072 p = 104857601 : 196.147 (us)
多项式乘法结果正确
average latency for n = 131072 p = 469762049 : 195.546 (us)
```

图 6.4: 三模数 CRT 动态线程运行结果

第二张图为为三模数 CRT 静态线程运行结果。

通过两张图片，我们可以看到静态线程的效率更高，原因如下：

```
[s2312632@master_ubss1 ntt]$ g++ main.cc -o main -O2 -fopenmp -lpthread -std=c++11 && ./main
多项式乘法结果正确
average latency for n = 4 p = 7340033 : 0.0929 (us)
多项式乘法结果正确
average latency for n = 131072 p = 7340033 : 194.07 (us)
多项式乘法结果正确
average latency for n = 131072 p = 104857601 : 192.067 (us)
多项式乘法结果正确
average latency for n = 131072 p = 469762049 : 192.165 (us)
```

图 6.5: 三模数 CRT 静态线程运行结果

静态线程在程序启动时一次性创建好固定数量的工作线程，后续所有的并行任务都在这些线程之间复用，无需重复进行线程的创建和销毁，从而消除了操作系统级别的资源分配与释放开销。此外，线程池中的线程长期驻留，利用条件变量或任务队列快速唤醒，可以实现低延迟的任务调度与上下文切换；相比之下，动态线程范式每次并行区域都要重新创建和销毁线程，不仅增加了调度延迟，还导致频繁的内核调度和资源管理开销。静态线程的常驻特性也有助于保持 CPU 缓存中的工作集，提高缓存命中率，而动态线程在频繁进出时则会引入较高的缓存暖启动成本。综上，静态线程通过复用线程资源、减少频繁调度与内存开销，实现了更稳定、更高效的并行执行。

而动态线程的优势在于动态线程按需创建与销毁，不会在并行任务稀疏或短暂时长期占用系统资源，能够更灵活地响应负载波动，降低空闲线程的内存和句柄占用，并在资源受限或任务不稳定的场景中实现更高的资源利用率和可伸缩性。

6.1.2 OpenMP 运行结果

和上面 pthread 两组进行对比，我发现 OpenMP 并行的效率更高，原因如下：

OpenMP 在编译期通过指令（pragma）直接生成并行代码，编译器能够对循环、数据访问和内存布局进行全局优化，并自动插桩以减少手动管理的开销；同时，OpenMP 运行时提供了线程池、负载均衡和任务调度的内置支持，使用工作窃取和动态调度策略来均匀分配计算任务，最大化缓存亲和性并减少同步等待，而 pthread 则需要程序员显式创建、销毁线程及管理同步原语，容易产生过度同步或负载不均的问题，导致更高的上下文切换和调度开销。

```
[s2312632@master_ubss1 ntt]$ g++ main.cc -o main -O3 -fopenmp -lpthread -std=c++11 && ./main
多项式乘法结果正确
average latency for n = 4 p = 7340033 : 0.560351 (us)
多项式乘法结果正确
average latency for n = 131072 p = 7340033 : 180.894 (us)
多项式乘法结果正确
average latency for n = 131072 p = 104857601 : 181.118 (us)
多项式乘法结果正确
average latency for n = 131072 p = 469762049 : 181.389 (us)
```

6.2 Profiling

6.2.1 Perf 多项指标分析

动态线程　下图展示了使用 `perf record` + `perf report` 对程序 `main` 采样分析后的热点函数分布：

```
Samples: 7K of event 'cycles:u', Event count (approx.): 4524621133
Overhead  Command  Shared Object       Symbol
  79.50%  main     libgcc_s-10.3.1.so.1  [.] __modti3
   5.31%  main     main                  [.] ntt
   5.27%  main     libstdc++.so.6.0.28   [.] std::num_get<char, std::istreambuf_iterator<char, std::char_traits<char> > >::_M_extract_int<long>
   2.50%  main     main                  [.] __modti3@plt
   1.26%  main     libstdc++.so.6.0.28   [.] std::istream::sentry::sentry
   0.90%  main     libstdc++.so.6.0.28   [.] 0x000000000011e578
   0.70%  main     libstdc++.so.6.0.28   [.] std::istream::operator>>
   0.50%  main     libstdc++.so.6.0.28   [.] std::__ostream_insert<char, std::char_traits<char> >
   0.46%  main     libstdc++.so.6.0.28   [.] std::basic_filebuf<char, std::char_traits<char> >::xsputn
   0.43%  main     libstdc++.so.6.0.28   [.] std::ostream::sentry::sentry
   0.32%  main     libstdc++.so.6.0.28   [.] std::ostream::_M_insert<long>
   0.27%  main     libstdc++.so.6.0.28   [.] std::num_put<char, std::ostreambuf_iterator<char, std::char_traits<char> > >::_M_insert_int<long>
```

表 2: Perf Report 分析结果

指标	分析结果
主要开销	`_modti3` (79.50%)：模运算开销极大；`main` (5.31%)：主程序逻辑有一定开销；`libstdc++` 函数 (2.77%-0.32%)：I/O 和字符串处理开销。
动态线程影响	运行时线程创建和销毁增加开销，I/O 操作可能导致线程阻塞和上下文切换。
I/O 操作	标准库函数（如 `std::num_get`, `std::ostream::sentry`）表明频繁 I/O 操作。
性能瓶颈	模运算、I/O 操作和动态线程管理开销是主要瓶颈。

动态 `pthread` 程序的性能瓶颈主要集中在模运算、I/O 操作和线程管理开销上。

下表展示了执行命令

```
perf stat -e cycles,instructions,cache-misses ./main
```

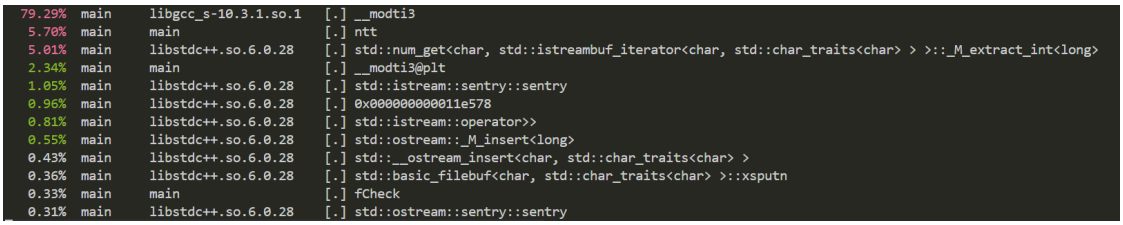
得到的关键性能计数器结果：

表 3: Perf Stat 分析结果

指标	分析结果
周期与指令	周期数：4,641,699,740；指令数：9,531,325,045；每周期指令数 (IPC)：2.05，指令流水线利用率较高。
缓存未命中	缓存未命中数：10,781,601；未命中率：0.113%，内存访问效率较高，但未命中绝对值仍较大。
时间分布	总时间：0.8561 秒；用户态：1.7848 秒；系统态：0.0995 秒，用户态时间远高于系统态，计算逻辑开销为主。
动态线程影响	用户态时间累加表明使用动态 `pthread`，线程创建和销毁可能增加调度开销。

程序的 IPC 为 2.05，缓存未命中率较低，但动态线程和计算逻辑仍为主要开销来源。

静态线程 这个是静态线程的`perf record` + `perf report` 对程序 `main` 采样分析后的热点函数分布：



静态 `pthread` 程序的性能分析表明，其性能瓶颈主要集中在以下几个方面：首先是模运算的高开销，尤其是 `_modti3` 函数占用高达 79.29% 的运行时间，表明 128 位整数取模操作是计算密集型任务的核心瓶颈；其次是 I/O 操作带来的额外开销，标准库函数如 `std::num_get` 和 `std::ostream::sentry` 等合计约占 9.71% 的性能，频繁的输入输出操作可能导致效率低下；最后是线程管理开销，静态 `pthread` 的线程创建、同步和销毁过程可能引入上下文切换和资源竞争，进一步加剧性能负担。这些因素共同限制了程序的整体性能，若要优化，需重点针对模运算效率、I/O 频率以及线程管理策略进行改进。

表 4: Perf Report 分析结果

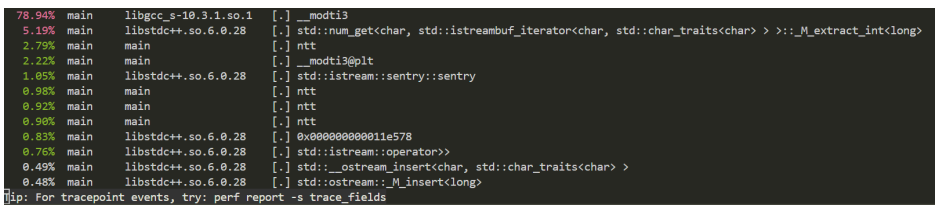
指标	分析结果
主要开销	`_modti3` (79.29%)：模运算开销占主导；`ntt` (5.70%)：NTT 核心逻辑有一定贡献；`libstdc++` 函数 (5.01%-0.31%)：I/O 和流处理开销。
动态线程影响	`pthread` 线程创建和销毁可能增加额外开销，I/O 操作可能引发线程阻塞和上下文切换。
I/O 操作	标准库函数（如 `std::num_get`, `std::istream::operator>>`, `std::ostream::sentry`）显示频繁的输入输出操作。
性能瓶颈	模运算 (`_modti3`)、I/O 操作和线程管理开销是主要瓶颈。

表 5: Perf Stat 分析结果

指标	分析结果
总运行时间	总计 0.838552177 秒，其中用户时间 1.813665000 秒，系统时间 0.044076000 秒，表明程序大部分时间在用户态执行，但系统调用开销也不可忽略。
CPU 效率	总周期数为 4,636,810,636，指令数为 9,531,325,136，平均每周期指令数（IPC）为 2.06，表明 CPU 利用率较高，但仍有优化空间。
缓存未命中	缓存未命中次数为 10,599,656，表明内存访问模式可能存在较大开销，数据局部性需进一步优化。
性能瓶颈	高缓存未命中率、模运算 (`_modti3`) 和 I/O 操作是主要瓶颈，pthread 线程管理可能进一步加重开销。

静态 `pthread` 程序的性能分析显示，其性能瓶颈主要集中在以下几个方面：首先，模运算的高开销（结合之前 perf report，`_modti3` 占 79.29%）是主要的计算瓶颈，可能由于频繁的 128 位整数取模操作；其次，高达 10,599,656 次的缓存未命中表明内存访问模式不够优化，数据局部性较差，导致频繁的内存访问延迟；此外，I/O 操作（结合之前报告，约占 9.71%）和 pthread 线程管理开销（如线程创建、同步和销毁）进一步加剧了性能负担。程序总运行时间为 0.838552177 秒，虽然 IPC 达到 2.06，显示 CPU 利用率较高，但缓存未命中和 I/O 操作的开销显著影响了整体效率。优化方向应包括改进模运算算法、优化数据访问模式以提升缓存命中率、减少不必要的 I/O 操作，并评估 pthread 线程管理策略以降低上下文切换和同步开销。

OpenMP 这个是 OpenMP 的`perf record` + `perf report` 对程序 `main` 采样分析后的热点函数分布：



OpenMP 并行版本的 `pthread` 程序性能分析表明，其瓶颈主要集中在以下几个方面：首先，模运算的高开销 (`_modti3` 占 78.94%) 依然是计算核心的瓶颈，表明并行并未有效减轻这一密集计算负担；其次，`ntt` 函数的多次出现 (总计约 5.59%) 且分布不均，暗示 OpenMP 并行中可能存在负载不平衡或线程同步开销，影响了并行效率；此外，I/O 操作 (约 6-7%) 由标准库函数如 `std::num_get` 和 `std::ostream::sentry` 贡献，表明并行执行中输入输出操作的开销未被优化，可能是串行 I/O 导致

的瓶颈；最后，结合之前的 perf stat 数据，高缓存未命中率和线程管理开销可能进一步加剧性能问题。优化方向包括：改进模运算算法以减少计算复杂度、优化 OpenMP 的线程分配以平衡负载、减少不必要的 I/O 操作，并评估数据局部性以降低缓存未命中率。

表 6: Perf Report 分析结果 (OpenMP 并行版本)

指标	分析结果
主要开销	`_modti3` (78.94%)：模运算仍占主导，可能与 NTT 核心计算相关；`std::num_get` (5.19%)：I/O 操作开销显著；`ntt` (2.79%-0.90%)：并行 NTT 逻辑分散但总体贡献较小。
并行影响	OpenMP 并行可能因线程同步或调度开销增加，而 `ntt` 函数的多次调用 (2.79%, 0.98%, 0.92%, 0.90%) 表明工作负载分配不均或存在负载不平衡。
I/O 操作	标准库函数（如 `std::num_get`, `std::istream::operator>>`, `std::ostream::sentry`）合计约 6-7%，显示并行中 I/O 仍为瓶颈。
性能瓶颈	模运算 (`_modti3`)、I/O 操作和 OpenMP 的线程同步/负载不平衡是主要瓶颈。

表 7: Perf Stat 分析结果

指标	分析结果
总运行时间	总计 0.785926249 秒，其中用户时间 1.783706000 秒，系统时间 0.085227000 秒，显示程序大部分时间在用户态执行，系统调用开销相对较小。
CPU 效率	总周期数为 4,655,833,917，指令数为 9,565,483,910，平均每周期指令数（IPC）为 2.05，表明 CPU 利用率较高，但略低于理想值，可能存在轻微的指令级并行限制。
缓存未命中	缓存未命中次数为 11,406,100，相比之前报告的 10,599,656 次略有增加，表明内存访问模式可能仍需优化，缓存利用率有待提升。
性能瓶颈	缓存未命中率较高、模运算 (`_modti3`) 和 I/O 操作是主要瓶颈，结合 OpenMP 并行可能存在线程同步或负载不平衡问题。

perf stat 报告显示，OpenMP 并行版本的 `pthread` 程序在本次运行中总耗时为 0.785926249 秒，相比之前 0.838552177 秒有所改善，表明可能存在一定优化效果。然而，性能瓶颈依然显著：首先，模运算 (`_modti3`，之前占 78.94%) 仍是核心计算负担，未见明显减轻；其次，缓存未命中次数增至 11,406,100，显示内存访问模式可能因并行数据分布不均而恶化，数据局部性需进一步优化；此外，IPC 为 2.05 虽较高，但与理想值相比仍有差距，可能受限于指令依赖或线程同步开销；最后，I/O 操作 (之前约 6-7%) 和系统调用 (0.085227000 秒) 也贡献了部分开销。优化建议包括：改进模运算算法以减少计算复杂度、调整 OpenMP 线程分配以平衡负载、优化数据布局以提升缓存命中率，以及减少不必要的 I/O 操作以降低系统开销。

6.3 一些个人的思考

基于提供的 perf report 和 perf stat 数据分析，模运算（___modti3）作为 NTT 的核心计算负担，在静态 pthread 版本中占比高达 79.29%，而在 OpenMP 并行版本中也达到 78.94%，其严重性显著限制了程序的整体性能。结合 perf stat 报告（OpenMP 版本总耗时 0.785926249 秒，静态版本 0.838552177 秒，缓存未命中分别为 11,406,100 次和 10,599,656 次）以及 OpenMP 中 ntt 函数的负载不平衡（2.79%-0.90%），实验表明模运算的高开销、线程管理开销和缓存未命中是主要瓶颈。

在继续使用 pthread 和 OpenMP 的基础上，可以采取以下改进措施以提升效率：首先，针对 pthread 的高系统调用开销（OpenMP 版本系统时间 0.085227000 秒），可以通过实现 pthread 线程池来预创建线程，避免线程频繁创建和销毁，减少上下文切换成本，从而提升线程复用效率。

其次，在 OpenMP 并行中，ntt 函数调用分布不均显示出负载不平衡问题，可通过使用 pragma omp parallel for schedule(dynamic, chunk) 动态调度（例如设置 chunk=100）来调整线程间的工作分配，确保计算负载更均衡。

此外，缓存未命中率较高的问题可以通过优化数据局部性来缓解，具体做法是将 NTT 输入数据分块并分配给不同线程，确保每个线程访问连续内存区域，从而降低缓存未命中率。

最后，I/O 操作（占比约 6-7%）对性能的影响不容忽视，可以在 pthread 中创建专用 I/O 线程，或者在 OpenMP 中使用 pragma omp sections 将 I/O 操作与计算任务分离，避免 I/O 阻塞计算线程。这些改进措施在保留 pthread 和 OpenMP 并行框架的基础上，能够有效缓解模运算负担、优化线程管理并提升整体性能，同时符合实验中观察到的性能瓶颈问题。

参考文献

[1] https://blog.csdn.net/weixi_44885334/article/details/134532078

[2] V4: The Number-Theoretic Transform (NTT) [Slide presentation]. © Alfred Menezes.