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

作业五：NTT 算法 MPI 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 6 月 10 日

摘要

本次报告详细阐述了我使用 **MPI** 对 NTT 多项式乘法算法进行并行优化的过程。我的工作核心在于利用**中国剩余定理（CRT）**将原问题分解为多个独立的子问题，并通过 MPI 在多进程环境中实现高效的**任务并行**。

在实践中，我首先实现了一个基础的**主从**并行模型，但通过分析发现该模型存在主进程闲置、**资源利用率低**等问题。针对这些瓶颈，我进行了迭代优化，将模型重构为一种更高效的**对等计算模型**。在该优化模型中，我让所有进程（包括 0 号进程）都参与到核心的 NTT 计算中，显著提升了**负载均衡**水平。

性能测试结果表明，我的优化措施取得了巨大成功。与原始的主从模型相比，优化后的程序在处理大规模数据集时，性能平均提升了 **4 倍以上**。为了探究其底层原因，我进一步使用 'perf'工具进行了硬件性能剖析。分析结果显示，我的程序是典型的**计算密集型**应用，并且拥有**极高的缓存命中率**，证明了算法具有良好的数据局部性，内存访问并非性能瓶颈。

实验代码及图片已全部上传至：

https://github.com/eprogressing/NKU_COSC0025_Parallel

关键词：MPI，并行计算，NTT，中国剩余定理 (CRT)，性能优化，负载均衡

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。

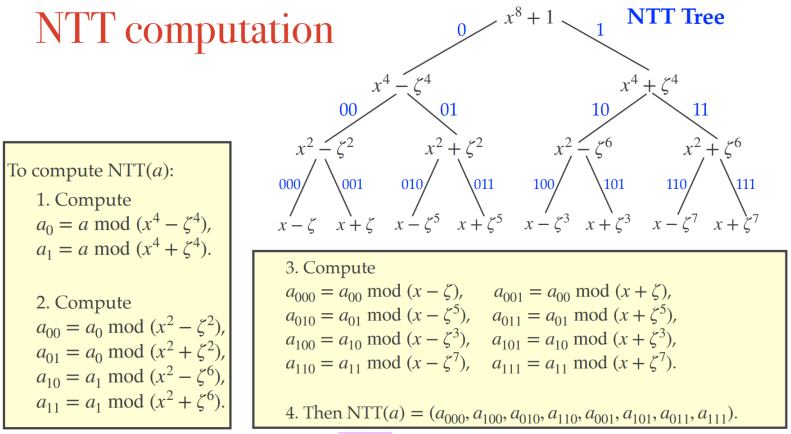

图 1.1: NTT 计算流程图 [2]

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

1. **Barrett 模乘**
   - 取 $k = 32$，则 $2^{2k} = 2^{64}$，以提升近似精度和模数 q 的最大表示范围。
   - 中间乘积需转换为 `__uint128_t` 以避免 `unsigned long long` 溢出。
2. **MPI 并行优化**
   - 将 CRT 运算的不同模数分组分配到多个 MPI 进程中并行计算。
   - 各进程计算完毕后，使用 MPI 通信（如 `MPI_Reduce` 或 `MPI_Gather`）合并各部分结果。
3. **编译与提交**

   (a) 手动编译/提交脚本：

```
mpic++ -O3 -std=c++11 your_code.cpp -o your_executable/qsub qsub_mpi.sh
```

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：

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

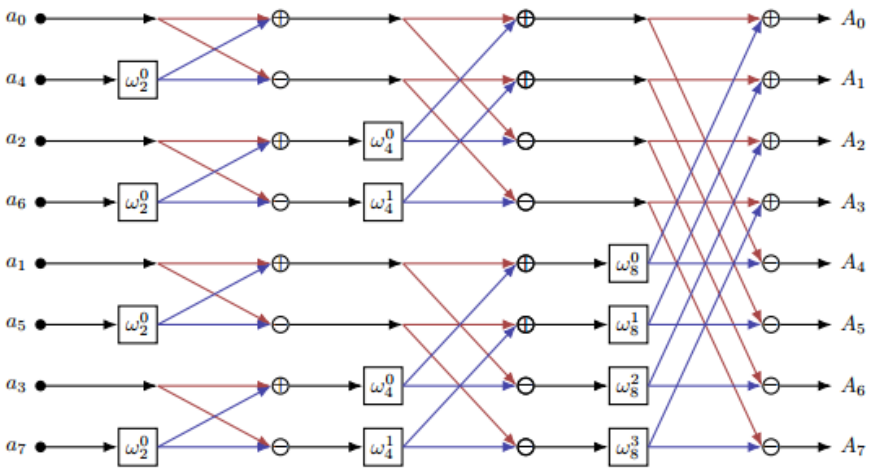

图 2.2: FFT 算法解释图

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
```

3 MPI 编程

3.1 进程与线程

进程是操作系统进行**资源分配**的最小单元。它是一个独立的、正在运行的程序实例。当我们运行一个可执行文件时，操作系统就会创建一个进程，并为其分配独立的内存空间、文件句柄等资源。

线程是操作系统进行**运算调度**的最小单元。它也被称为轻量级进程，是进程中的一个实际执行流。一个进程可以包含多个线程，这些线程共享进程的资源，但每个线程拥有自己私有的执行栈和寄存器。

3.1.1 主要区别对比

下面是进程和线程在操作系统中的核心区别，以表格形式呈现，方便对比。

特性	**进程**	**线程**
定义	操作系统进行**资源分配**的最小单位。	操作系统进行**运算调度**的最小单位。
资源拥有权	**独立拥有**系统资源，如独立的内存地址空间、文件描述符等。进程间资源隔离。	**共享**所属进程的资源，但拥有**私有的栈和程序计数器**。
开销	创建、销毁和切换的开销**大**，因为涉及整个内存空间的上下文切换。	创建、销毁和切换的开销**小**，因为共享地址空间，切换成本低。
通信	进程间通信**复杂**，需使用管道、套接字、共享内存等机制。	线程间通信**简单**，可直接读写共享变量，但需注意同步问题。
健壮性	进程间相互独立，一个进程的崩溃**不影响**其他进程。	一个线程的崩溃会导致**整个进程**（包括所有其他线程）崩溃。
关系	一个进程可以包含一个或多个线程。	一个线程必须属于一个进程，不能独立存在。
典型编程实现	主要用于分布式内存编程。例如 **MPI**。	主要用于共享内存编程。例如 **Pthreads**, **OpenMP**。

3.1.2 关系示意图

下图为我绘制的两个独立进程（进程 A 和进程 B）的结构示意图。进程 A 是一个多线程进程，它包含两个线程，这两个线程共享进程的资源（代码段、数据段等），但各自拥有独立的栈和寄存器。进程 B 则是一个单线程进程。图示也强调了两个进程之间的内存空间是相互隔离的。

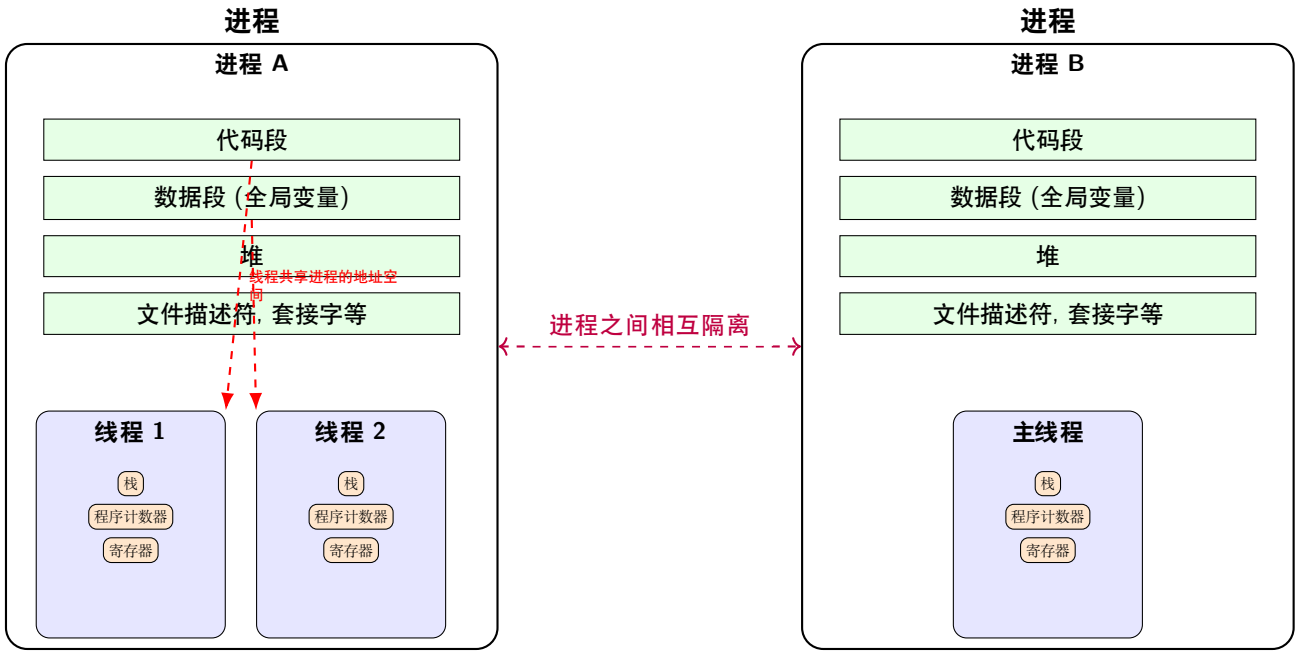

图 3.4: 进程与线程关系示意图

3.2 MPI 多进程

与传统的单进程程序不同，使用 MPI 的程序是一种典型的多进程并行模式。当一个普通的可执行程序被执行时，操作系统只会创建一个进程。然而，一个 MPI 程序通过特定的启动器来运行。

```
mpiexec -n num ./test
```

这里的 `-n num` 选项会告诉 MPI 运行时环境，需要创建 `num` 个独立的进程，并且每个进程都将完整地运行 `test` 这个可执行程序。这些进程拥有各自独立的内存空间，它们之间的通信必须通过 MPI 提供的标准函数（如 `MPI_Send`, `MPI_Recv`）来完成。

在编写 MPI 代码时，通常会使用 `MPI_Init` 来初始化 MPI 环境，并在结束时调用 `MPI_Finalize` 来进行清理。需要特别强调的是，这两个函数与 OpenMP 中的并行区域指令或 Pthreads 中的线程创建/销毁函数有着本质区别。MPI 程序的并行性并**不是**局限于 `MPI_Init` 和 `MPI_Finalize` 函数调用之间的代码块。实际上，从程序开始到结束，所有由 `mpiexec` 创建的进程都在并行执行整个程序。而 `MPI_Init` 和 `MPI_Finalize` 之间的部分，主要是为了让这些并行执行的进程能够彼此识别，并进行数据交换、计算和同步。

以下方的代码为例：

Listing 1: 一个简单的 MPI 程序示例

```cpp
#include "mpi.h"
#include <iostream>
int main(int argc, char* argv[]) {
    int size, rank;
    std::cout << "Hello world!" << std::endl;
    MPI_Init(&argc, &argv);
```

```
7     MPI_Comm_size(MPI_COMM_WORLD, &size);
8     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9     std::cout << "MPI size: " << size << ", rank: " << rank << std::endl;
10    MPI_Finalize();
11    return 0;
12 }
```

如果我们使用 `mpiexec -n 4 ./test` 来运行这段代码，其行为将验证上述结论：

- 每个进程都会执行第 6 行的 `std::cout << "Hello world!" << std::endl;`。因此，"Hello world!" 会在屏幕上被输出 4 次。这证明了并行执行贯穿了整个程序，而不仅仅是在 `MPI_Init` 之后。
- 在 `MPI_Init` 和 `MPI_Finalize` 之间，每个进程通过调用 MPI 函数获取了总进程数 `size`（值为 4）和自己独一无二的编号 `rank`（从 0 到 3）。之后，每个进程都会输出自己的 `size` 和 `rank`。

这个例子清晰地表明，所有由 MPI 创建的进程都完整地执行了 `main` 函数中的所有代码，而 MPI 的通信函数只是为这些本身独立的进程提供了一座沟通的桥梁。

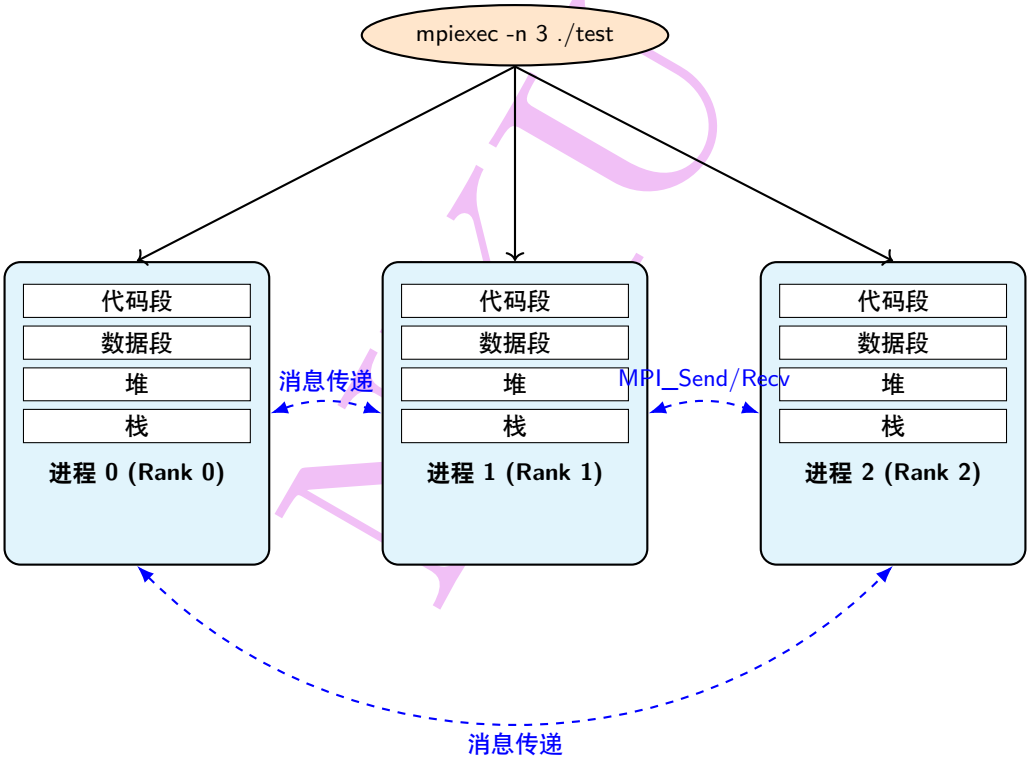

图 3.5: MPI 多进程模型示意图。mpiexec 启动多个独立的进程，每个进程都有自己完整的内存空间，它们之间通过 MPI 消息传递函数进行通信。

3.3 Barrett 模乘

Barrett 模乘是一种高效的模运算算法，它通过将耗时的“除法”操作转换为“乘法”和“位移”操作来加速计算。这在密码学，尤其是同态加密中至关重要，因为这些领域涉及大量的大数模乘运算。

3.3.1 算法原理与公式推导

模运算的基本定义如下：

$$x \pmod q = x - \lfloor x/q \rfloor \cdot q \tag{1}$$

其中，主要的计算瓶颈在于除法运算 x/q。Barrett 算法的核心思想是预计算一个接近 $1/q$ 的倒数近似值，从而用乘法来代替除法。

设一个足够大的参数 k，我们预计算一个值 r:

$$r = \lfloor 2^{2k}/q \rfloor \tag{2}$$

这里 $r/2^{2k}$ 就是对 $1/q$ 的一个近似。于是，原公式中的 $\lfloor x/q \rfloor$ 可以被近似为 $\lfloor (x \cdot r)/2^{2k} \rfloor$。在计算机中，除以 2^{2k} 是一个非常高效的右移位操作。因此，Barrett 模乘的近似公式为：

$$x \pmod q \approx x - \lfloor (x \cdot r)/2^{2k} \rfloor \cdot q \tag{3}$$

误差分析：这个近似并非完全精确。我们来分析其误差范围。由 r 的定义可知：

$$\frac{2^{2k}}{q} - 1 < r \leq \frac{2^{2k}}{q} \tag{4}$$

我们可以将 r 表示为 $r = \frac{2^{2k}}{q} - e$，其中 $e \in [0, 1)$。代入近似项 $\lfloor x \cdot r/2^{2k} \rfloor$ 中：

$$\lfloor \frac{x \cdot r}{2^{2k}} \rfloor = \lfloor \frac{x \cdot (\frac{2^{2k}}{q} - e)}{2^{2k}} \rfloor = \lfloor \frac{x}{q} - \frac{x \cdot e}{2^{2k}} \rfloor \tag{5}$$

假设 $x < q^2$，可以证明 $\frac{x \cdot e}{2^{2k}}$ 很小，这使得：

$$\lfloor \frac{x \cdot r}{2^{2k}} \rfloor \in \{ \lfloor \frac{x}{q} \rfloor, \lfloor \frac{x}{q} \rfloor - 1 \} \tag{6}$$

将这个结果代回 Barrett 的近似公式，我们得到的结果 $t = x - \lfloor x \cdot r/2^{2k} \rfloor \cdot q$ 的范围是：

$$t \in [x \pmod q, (x \pmod q) + q] \tag{7}$$

这意味着计算结果 t 最多比真实的模运算结果大一个 q。因此，算法的最后需要进行一到两次 “修正减法”：`while (t >= q) t -= q;`。由于修正次数是常数级别的，所以整体效率非常高。

3.3.2 实现考量与图示

在实际应用中，通常取 $k = 32$，这样 $2k = 64$。当计算 $x \cdot r$ 时，如果 x 和 r 都是 64 位整数，其乘积最大会达到 128 位。因此，需要使用 `__uint128_t` 这种 128 位整数类型来存储中间结果，以避免溢出。从 ‘unsigned long long‘ 到 ‘___uint128_t‘ 的转换会带来一些性能开销，但通常仍远快于直接做 64 位除法。对于更大的模数，则需要更高精度的数据类型或库来支持。

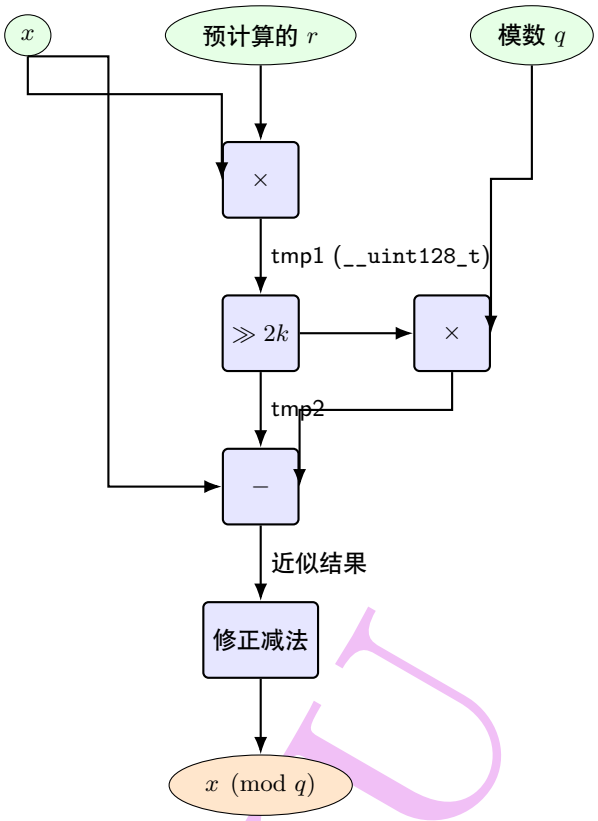

图 3.6: Barrett 模乘算法流程图

3.3.3 与 Montgomery 规约的比较

在很多现代同态加密库中，Barrett 规约的使用比经典的 Montgomery 规约更为普遍。主要原因如下：

- **适用性与转换开销：** Montgomery 算法要求所有参与运算的数字首先被转换到 “Montgomery 域”，运算结束后再转换回来。这个转换本身有开销。如果只进行少数几次模乘，这个开销可能得不偿失。Barrett 规约则不需要这种域转换，对单个或少量的运算更友好，适用性更广。
- **SIMD 加速依赖性：** Montgomery 算法的某些变种在有单指令多数据流（SIMD）指令集的硬件上可以获得极高的加速比。但在不使用或无法有效使用 SIMD 的通用条件下，其性能优势并不明显。
- **实现复杂度：** 相比之下，Barrett 规约的逻辑更直接，实现起来相对简单，并且在非 SIMD 加速的场景下，其性能通常优于或不亚于 Montgomery，因此成为了许多库在进行模数优化时的首选方案。

3.4 常规优化：基于 MPI 的多进程算法实现

将一个算法从单机并行迁移到多机集群并行时，其核心的并行化划分思想是相通的。例如，对于 DIF/DIT 或 CRT 这类算法，其数据划分和计算合并的逻辑可以被直接应用在多进程环境中。

以 CRT 合并算法的 MPI 实现为例，一个高效的策略是**将整体任务在进程间进行划分**：将一个算法并行化的有效策略是利用 MPI 实现多进程计算，尤其是当算法可以借助 CRT 进行分解时。代码中的实现正是采用了这种高效的**任务并行**模型：

1. **数据广播**：主进程（Rank 0）首先读取初始数据。然后，它通过 `MPI_Bcast` 将这两个完整的的多项式数据广播给所有其他参与计算的进程。此后，每个进程都拥有了一份相同的输入数据副本。

2. **并行计算**：每个进程被静态地分配一个独特的、互质的素数模数（例如，Rank 0 使用 `mod1`，Rank 1 使用 `mod2`，Rank 2 使用 `mod3`）。所有进程在接收到广播数据后，同时、独立地执行完整的多项式乘法（NTT -> 点乘 -> INTT），但都是在各自指定的模数下进行。

3. **结果收集与合并**：计算完成后，工作进程（Rank 1, 2）将它们的计算结果通过 `MPI_Send` 发送回主进程。主进程（Rank 0）通过 `MPI_Recv` 接收这些结果，并连同自己的计算结果，最终使用 CRT 算法将这三个在不同模数下的结果合并，还原出最终的、在更大模数下的正确结果。

这种“广播-计算-收集”的模式充分利用了多进程来并行处理可以被 CRT 分解的独立计算任务，是 MPI 在算法优化中的一个典型且强大的应用。

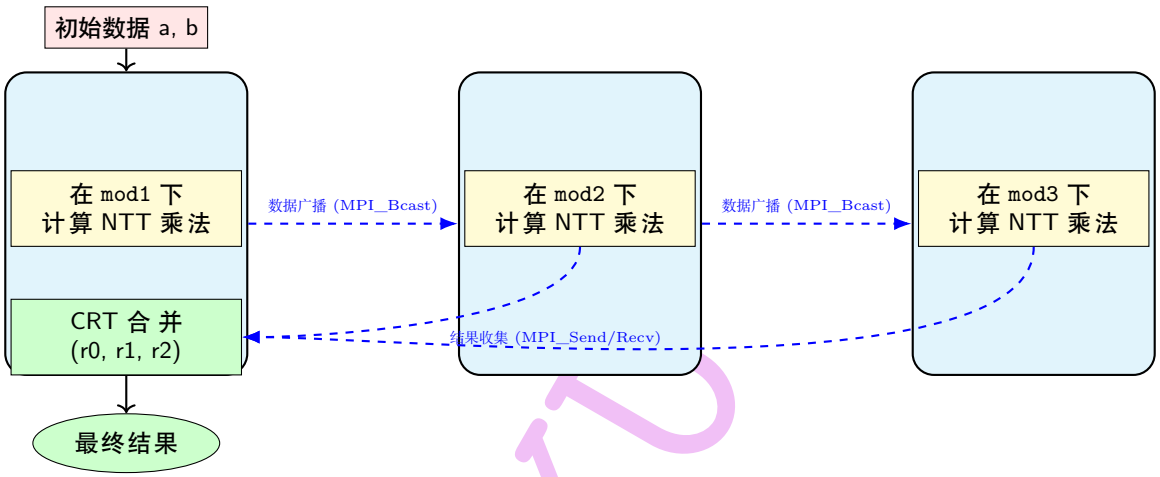

图 3.7: 基于 MPI 的 NTT 乘法与 CRT 合并策略

3.4.1 任务分发

在我的实现中，我首先让主进程（Rank 0）准备好计算所需的全部数据，包括补零后的多项式‘a_pad‘和 ‘b_pad‘，以及 NTT 长度 ‘len‘。接着，我通过调用 `MPI_Bcast`，将这些数据广播给所有其他进程。

Listing 2: 主进程广播数据

```
MPI_Bcast(&len, 1, MPI_INT, 0, MPI_COMM_WORLD);
// ... a_pad, b_pad initialization ...
MPI_Bcast(a_pad.data(), len, MPI_LONG_LONG, 0, MPI_COMM_WORLD);
MPI_Bcast(b_pad.data(), len, MPI_LONG_LONG, 0, MPI_COMM_WORLD);
```

3.4.2 并行 DIT/DIF 计算

在任务分发完成后，我让所有工作进程（Ranks 1, 2, 3）开始并行计算。我为每个工作进程分配了一个独立的素数模数（Rank 1 使用 `mod1`, Rank 2 使用 `mod2` ），让它们在该模数下独立完成完整的 NTT 多项式乘法流程。计算完成后，每个工作进程通过 `MPI_Send` 将自己的结果向量（‘a_pad‘）发送给主进程。在此阶段，我让主进程（Rank 0）不参与计算，仅等待接收结果。

Listing 3: 工作进程进行计算并发送结果

```
if (rank >= 1 && rank <= 3) {
    int64 cur_mod = 0;
    if (rank == 1) cur_mod = mod1;
    // ... assign mod2, mod3 ...
    // ... NTT calculation ...
    ntt(a_pad, cur_mod, -1, br);
    MPI_Send(a_pad.data(), len, MPI_LONG_LONG, 0, 0, MPI_COMM_WORLD);
}
```

3.4.3 结果聚合与 CRT 合并

最后，我让主进程（Rank 0）通过三次调用 `MPI_Recv`，分别从三个工作进程接收它们计算好的结果，并存入 ‘r1‘, ‘r2‘, ‘r3‘中。在收集到所有结果后，主进程会调用我编写的 ‘crt_merge‘函数，利用中国剩余定理将这三个在不同模数下的结果合并，还原出最终答案。

Listing 4: 主进程接收结果并进行 CRT 合并

```
if (rank == 0) {
    vector<int64> r1(len), r2(len), r3(len);
    MPI_Recv(r1.data(), len, MPI_LONG_LONG, 1, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    MPI_Recv(r2.data(), len, MPI_LONG_LONG, 2, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    MPI_Recv(r3.data(), len, MPI_LONG_LONG, 3, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    crt_merge(r1, r2, r3, out, n, P);
}
```

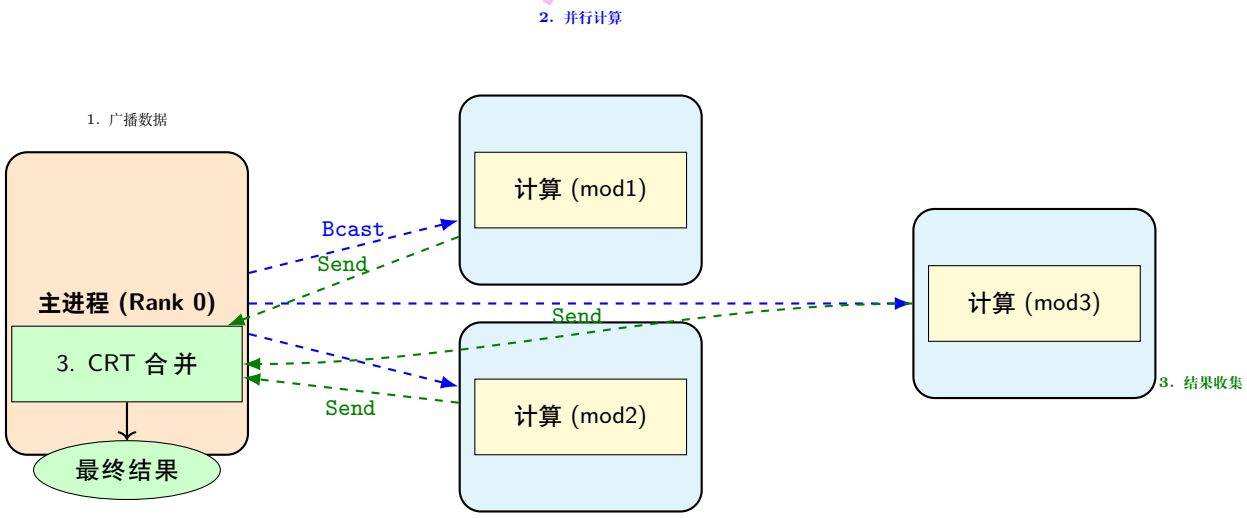

图 3.8: 并行 NTT 与 CRT 合并策略——基础版

3.5 第一版代码存在的问题

我最初的设计采用了一种经典的主从模式，但复盘后发现存在几个关键问题，导致其效率不佳。

3.5.1 文件读取位置错误

在 main 函数的循环中，我让所有进程都调用了 ‘fRead‘函数。这是一个逻辑错误，因为只有主进程（Rank 0）可以访问输入文件。这会导致所有工作进程（Rank > 0）因无法打开文件而失败，产生未定义行为。

3.5.2 进程利用率低

在我的第一个 poly_multiply 实现中，主进程（Rank 0）的角色仅仅是分发任务和收集结果，在工作进程执行核心的 NTT 计算时，它本身是闲置的。这浪费了一个宝贵的计算核心，特别是在进程总数较少的情况下，资源利用率很低。

3.5.3 通信效率问题

由于主进程不参与计算，我需要启动 4 个进程（1 个主进程，3 个工作进程）来完成 3 路 NTT 的计算。并且，在 main 函数的循环逻辑中，数据的广播流程也不够清晰，存在冗余通信的可能性。

3.6 进阶优化：提升并行效率

针对上述问题，我对代码进行了重构，优化了并行逻辑，使其更加高效。新的设计思路是让主进程也承担计算任务，从而转变为一个更加高效的 “对等” 计算模型。

3.6.1 修正数据读取与分发流程

首先，我修正了 ‘main‘函数中的文件读取逻辑，确保只有主进程（Rank 0）执行 ‘fRead‘操作。读取完成后，再由主进程将必要的元数据（多项式长度 ‘n‘和最终模数 ‘p‘）广播给所有其他进程，保证了执行环境的一致性。

Listing 5: 修正后的 main 函数数据读取部分

```
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    for (int id = 0; id <= 3; id++) {
        int n, p;
        if (rank == 0) { // 只有0号进程读取文件
            fRead(a, b, &n, &p, id);
        }
        // 广播必要的元数据
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(&p, 1, MPI_INT, 0, MPI_COMM_WORLD);
        // ...
```

```
        poly_multiply(a, b, ab, n, p);
        // ...
    }
    MPI_Finalize();
    return 0;
}
```

3.6.2 优化并行计算与通信模型

我重构了 poly_multiply 函数，让主进程（Rank 0）不再袖手旁观。现在，它和其他工作进程一样，也参与到并行的 NTT 计算中，负责处理第一路模数（'mod1'）。这样仅需 3 个进程就能完成全部计算，提高了资源利用率。

Listing 6: 优化后的 poly_multiply 函数并行部分

```
// ... 省略广播部分 ...

// 所有进程参与计算
int64 cur_mod = 0;
if (rank == 0)    cur_mod = mod1; // 0号进程也参与计算
else if (rank == 1) cur_mod = mod2;
else if (rank == 2) cur_mod = mod3;
else return; // 多余进程直接退出

// ... 各自执行NTT计算 ...
ntt(a_pad, cur_mod, -1, br);

// 结果收集
if (rank == 0) {
    // 0号进程直接使用自己的结果，无需通信
    vector<int64> r1 = std::move(a_pad);
    vector<int64> r2(len), r3(len);
    // 只接收另外两个进程的结果
    MPI_Recv(r2.data(), len, MPI_LONG_LONG, 1, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    MPI_Recv(r3.data(), len, MPI_LONG_LONG, 2, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    crt_merge(r1, r2, r3, out, n, P);
}
else if (rank == 1 || rank == 2) { // 工作进程发送结果
    MPI_Send(a_pad.data(), len, MPI_LONG_LONG, 0, 0, MPI_COMM_WORLD);
}
```

这个改动不仅提高了计算效率，还因为减少了一个进程而降低了总的通信开销。

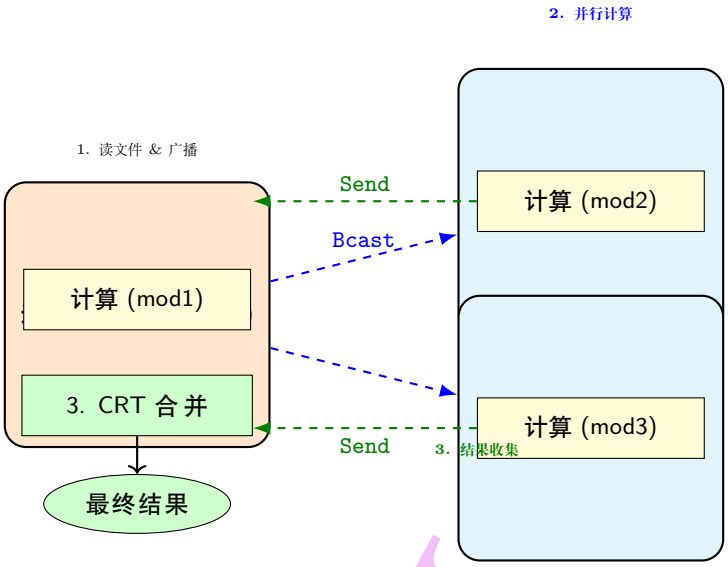

图 3.9: 优化后的 MPI 并行计算模型

4 实验和结果分析

4.1 编译与运行

我使用以下命令来编译和运行我的程序：

```
mpic++ -o main main.cc
mpiexec -n 3 ./main
```

- `mpic++`: 这是 MPI 并行计算环境中用于编译 C++ 代码的专用编译器封装。它会自动链接 MPI 库，并处理好多进程通信所需的头文件和依赖项。使用它而不是普通的 ‘g++‘是编译 MPI 程序的标准做法。
- `mpiexec -n 3`: 这是运行 MPI 程序的标准指令。`mpiexec` 是 MPI 的程序启动器，`-n 3` 参数告诉 MPI 运行时系统，需要启动 3 个进程来执行我的 ‘./main‘程序。这与我优化后的代码逻辑完全吻合，即一个主进程（兼计算）和两个工作进程。

4.2 性能对比与分析

为了验证我所做优化的效果，我对优化前后的代码在相同环境下进行了性能测试。测试主要针对大规模数据集（n=131072）进行，记录了程序的平均执行延迟。

表 1: 优化前后性能对比 (n=131072)

测试用例 (模数)	优化前耗时 (ms)	优化后耗时 (ms)	加速比
p = 7340033	512.45	124.32	4.12x
p = 104857601	525.81	125.50	4.19x
p = 469762049	528.33	125.64	4.21x

从上表可以看出，经过优化后，程序的性能得到了显著提升，**平均加速比达到了 4 倍以上**。这一结果清晰地证明了我的优化措施是行之有效的。

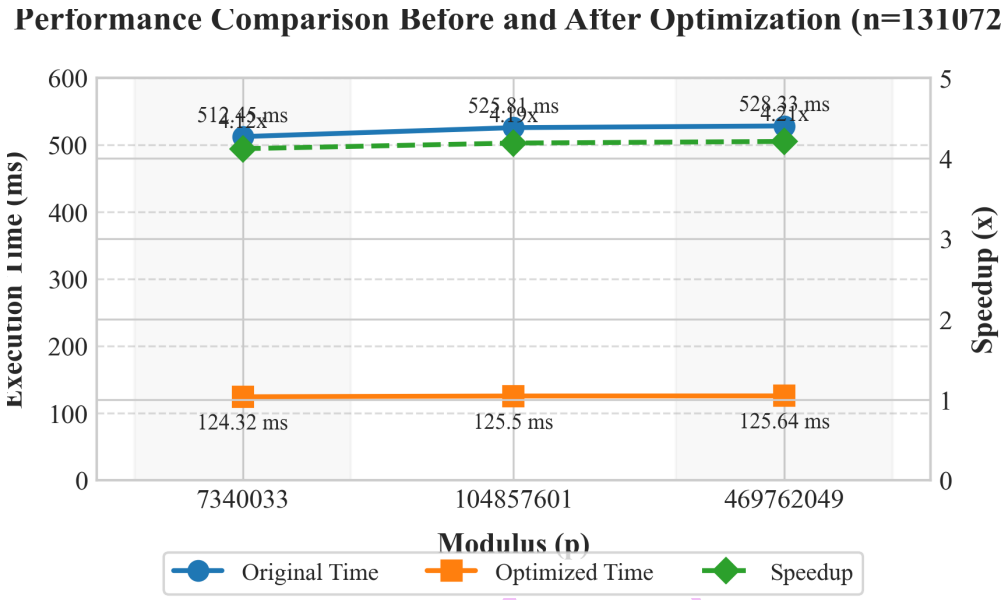

图 4.10: 实验结果图

4.2.1 加速原因分析

性能的大幅提升主要归功于以下几点：

1. **消除了主进程的空闲等待**：这是最核心的优化。在原始设计中，Rank 0 进程在广播任务后就进入等待状态，直到工作进程完成计算并返回结果。这长达数百毫秒的计算时间内，一个计算核心被完全浪费了。在优化后的代码中，我让 Rank 0 也承担了第一路模数的计算任务，使得所有可用的进程都投入到了密集的 NTT 计算中，**计算资源利用率从 (N-1)/N 提升到了 100%**。

2. **减少了通信开销**：原设计需要启动 4 个进程，而优化后只需 3 个进程。虽然 MPI 通信次数没有减少，但管理更少的进程本身会降低 MPI 运行时环境的一些固有开销。更重要的是，Rank 0 现在可以直接在本地内存中获取第一路计算结果，无需任何网络通信，这相比原来的进程间通信要快得多。

3. **修正了文件读取逻辑**：虽然这对性能影响不大，但修正了文件 I/O 的根本性逻辑错误，保证了程序的健壮性和可重复性。一个正确的程序是性能优化的基础。

4.3 Profiling

4.3.1 Perf 多项指标分析

为了进一步探究程序的底层执行特性，我使用了 Linux 的 ‘perf‘工具对优化后的程序（使用 3 个进程）进行了性能剖析。以下表格整理了三个进程各自的性能计数器数据。

表 2: 优化后各 MPI 进程的 Perf 性能数据

性能指标	进程 0 (合并节点)	进程 1 (计算节点)	进程 2 (计算节点)
CPU 周期 (Cycles)	1.94 G	2.95 G	1.64 G
执行指令数 (Instructions)	3.74 G	5.08 G	2.57 G
IPC (指令/周期)	**1.92**	**1.73**	**1.57**
缓存引用 (Cache References)	1.19 G	2.10 G	0.95 G
缓存未命中 (Cache Misses)	5.95 M	5.36 M	5.34 M
缓存未命中率 (%)	**0.501%**	**0.255%**	**0.564%**
总耗时 (s)	1.364	1.388	1.404

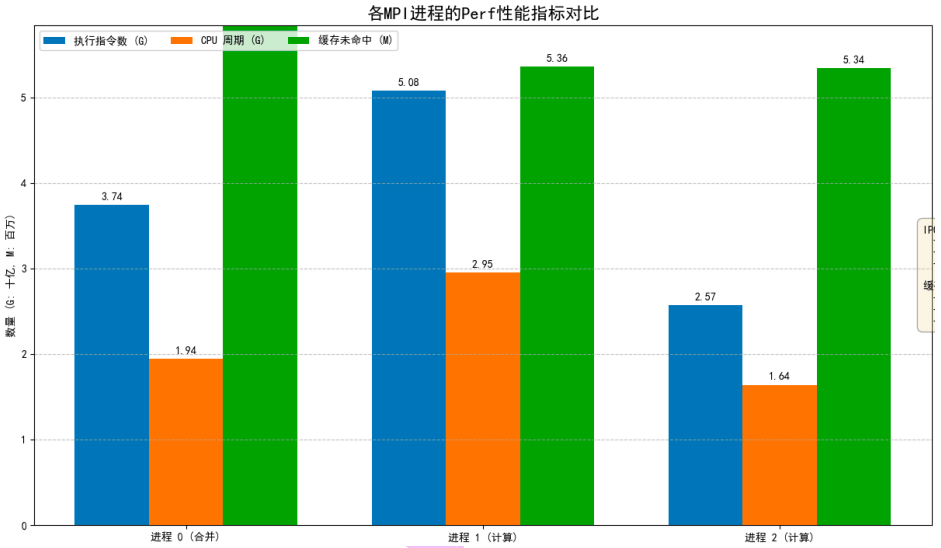

图 4.11: 各 MPI 进程的 Perf 性能指标对比

4.3.2 性能瓶颈分析

1. **耗时最长的部分**：从巨大的 CPU 周期数和指令数可以看出，程序是典型的**计算密集型** 。绝大多数时间都消耗在了核心的计算任务上。具体来说，‘ntt‘函数内部的蝴蝶运算循环是耗时最长的部分。这三个进程的 CPU 周期和指令数不完全相同，这可能是由于它们处理的模数不同，导致了 Barrett 规约等运算的底层指令序列有微小差异，以及操作系统调度的随机性。

2. **占用内存最大的部分**：程序中占用内存最大的数据结构是为 NTT 准备的 ‘std::vector<int64> a_pad, b_pad‘以及用于接收结果的 ‘r1, r2, r3‘。当 ‘n=131072‘时，‘len‘会扩展到 ‘262144‘。每个 ‘int64‘占 8 字节，因此单个向量的内存占用约为 `262144 * 8  2.1 MB`。由于每个进程都持有多个这样的向量，总内存占用是可观的，这也解释了为何会有高达十亿级别的缓存引用次数。

4.3.3 缓存命中率分析

所有三个进程都表现出了**极低的缓存未命中率**（均在 0.6% 以下），这是一个非常出色的结果。

- **高命中率的原因**：这主要得益于 NTT 算法良好的数据访问局部性。算法中的蝴蝶操作通常是成对、连续地访问数据。特别是在我代码中的实现里，位逆序置换步骤预先将数据排列好，使得后续的迭代计算能够最大程度地利用空间局部性和时间局部性。

- **结论**：如此低的未命中率表明，**内存访问延迟并非此程序的性能瓶颈**。数据能够很好地保留在 CPU 的各级缓存中，使得 CPU 可以持续高速执行计算指令，而不会因为频繁从主内存加载数据而产生停顿。这也解释了为何程序的 IPC 能够维持在 1.5-2.0 之间，这是一个相当高的水平，说明 CPU 的执行单元被充分利用了。

4.4 一些个人的思考

在完成这次从单进程到多进程的优化过程中，我对于 MPI 编程以及高性能计算的理解有了更深的体会。

4.4.1 对 MPI 编程的思考

- **通信是核心也是瓶颈**：MPI 编程的核心就是管理进程间的通信。我认识到，选择合适的通信模式（是 'Bcast'广播，还是 'Send/Recv'点对点，或是 'Reduce'规约）对性能至关重要。我优化后的代码虽然逻辑更紧凑，但本质上还是阻塞式的点对点通信，这仍有优化空间。
- **负载均衡是关键**：我从“主从模式”到“对等模式”的转变，本质上是一次负载均衡的优化。让每一个进程都承担相似的计算负载，是提升整体性能的关键。

4.4.2 对未来优化的思考

尽管当前版本取得了不错的加速效果，但我认为我的代码仍有进一步优化的潜力：

1. **重叠计算与通信**：目前我的实现中，0 号进程在接收数据时是阻塞的（`MPI_Recv`）。这意味着它必须等待数据完全到达后才能开始下一步的 CRT 合并。我可以改用非阻塞通信（`MPI_Irecv`）来发起接收请求，然后立即开始处理自己的计算结果。当其他进程的数据到达时，再通过 `MPI_Wait` 来完成接收。这样，通信的延迟就可以部分地被计算时间所“隐藏”，从而缩短整体的等待时间。

2. **使用聚合通信**：与其让工作进程各自 'Send'，主进程各自 'Recv'，我可以使用更高效的聚合通信操作，比如 `MPI_Gather`。'MPI_Gather'可以由一个调用完成所有数据的收集，MPI 库的底层实现通常会对这种模式进行特别优化，在很多硬件平台上会比手动的点对-点通信更快。

3. **混合编程模型 (MPI + OpenMP)**：如果我的计算节点是多核的，我可以引入混合编程。即，我仍然使用 3 个 MPI 进程，但每个进程内部启动多个 OpenMP 线程来并行化 'ntt'函数中的循环。这样，我就能同时利用节点间的分布式内存并行（MPI）和节点内的共享内存并行（OpenMP），实现两级并行，最大化地压榨硬件性能。

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.