南開大學
Nankai University

计算机学院

并行程序设计第五次报告

NTT 为例的 MPI 编程

姓名：宋卓伦

学号：2311095

专业：计算机科学与技术

2025 年 6 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 **2**
1.1 实验目的 . . . 2
1.2 实验硬件和环境 . . . 2

2 实验设计：MPI 的设计 **2**
2.1 MPI 编程简介 . . . 2
2.2 朴素 NTT 的 MPI 程序设计 . . . 3
2.3 算法简介 . . . 6

3 实验设计：与多线程以及 SIMD 的结合 **6**
3.1 与 OpenMP 的结合 . . . 7
3.2 与 Pthread 的结合 . . . 7
3.3 与 SIMD 的结合 . . . 7

4 实验设计：MPI 并行化不同算法策略 **8**
4.1 巴雷特规约优化 . . . 8
4.2 常规的四分优化 . . . 9

5 实验设计：MPI 不同的编程方法 **10**
5.1 阻塞通信 VS 非阻塞通信 . . . 10
5.2 单边通信 VS 多边通信 . . . 11
5.3 MPI 自身的多线程支持 . . . 12

6 程序性能分析 **12**
6.1 不同并行化方式的对比 . . . 13
6.2 对线程数的讨论 . . . 14
6.3 不同规约优化方法下的对比 . . . 14
6.4 对 OpenMP 的讨论 . . . 15

7 Profiling **16**
7.1 数据的总览 . . . 16
7.2 进一步的解释 . . . 17
7.3 数据分析 . . . 18

8 实验总结 **19**

A 实验的表格 **19**

1 实验目的及实验介绍

1.1 实验目的

通过课上的学习，我对 MPI 上的编程有了一定的了解和体会，在这里我将 MPI 编程尝试用在我们之前的 NTT 选题上，观察 MPI 并行化操作对我们的实验有什么影响，同时开始尝试不同并行化操作的结合与初步创新，为最后的 NTT 并行化算法总结进行一个铺垫。

1.2 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 服务器结合 MPI 专用编译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 实验设计：MPI 的设计

2.1 MPI 编程简介

MPI (Message Passing Interface)，即**消息传递接口**，是一种用于编写**并行计算程序**的标准库规范。它不是一种编程语言，而是一套函数库（API），允许程序员在分布式内存系统上协调多个独立进程的计算。在分布式内存系统中，每个处理器（或计算节点）都有自己的本地内存，并且不能直接访问其他处理器的内存。为了在这些处理器之间共享数据或协调工作，它们必须通过网络发送和接收消息。

MPI 的优势

- **扩展性**：支持数千至数万处理器，适合大规模并行应用。
- **高性能**：消息传递明确，底层优化确保高效通信。
- **灵活性**：提供点对点和集合通信，支持复杂并行算法。
- **可移植性**：开放规范（如 Open MPI, MPICH），代码跨平台可移植。

主要特性和概念

- **进程 (Rank)**：每个进程有唯一秩（0 到 `size-1`）。
- **通信器**：定义进程组，如 `MPI_COMM_WORLD`。

- **点对点通信**：`MPI_Send()` 和 `MPI_Recv()` 用于双进程数据传输。
- **集合通信**：
  - `MPI_Bcast()`：广播数据。
  - `MPI_Reduce()` / `MPI_Allreduce()`：数据聚合。
  - `MPI_Gather()` / `MPI_Scatter()`：数据收集/分散。
  - `MPI_Alltoall()`：全对全通信。
- **初始化和终止**：`MPI_Init()` 和 `MPI_Finalize()`。
- **环境变量**：通过 `mpirun -np N` 启动程序。

基本流程

1. 包含头文件：`#include <mpi.h>`
2. 初始化：`MPI_Init(&argc, &argv);`
3. 获取进程信息：
   - 秩：`MPI_Comm_rank(MPI_COMM_WORLD, &rank);`
   - 进程数：`MPI_Comm_size(MPI_COMM_WORLD, &size);`
4. 并行逻辑：基于 `rank` 和 `size` 分配任务，包括数据划分、计算、通信和结果收集。
5. 终止：`MPI_Finalize();`

MPI 的核心理念是**消息传递**。这意味着并行程序中的每个独立执行单元（通常称为一个“进程”或“任务”）都通过明确地发送和接收消息来与其他进程通信。每个进程都在自己的内存空间中操作，并且只通过这种消息传递机制来与其他进程的数据进行交互。

以上就是我们对于 MPI 编程一个详细的介绍。有了这些详细的了解，我们在后面的编程实现中才会更加有优势。

2.2 朴素 NTT 的 MPI 程序设计

我们结合之前朴素的 NTT 算法，插入 MPI 化的策略，进行实验。这里核心的要点在于在程序合适的地方添加 MPI 进程和线程的控制。

前面导入 mpi.h 头文件无需多言，是基本的操作，接下来我们直接描述程序具体过程：

进程信息 首先是获取进程信息：

```
int rank, size;
MPI_Comm_rank(comm, &rank);
// 获取当前进程在通信器 comm 中的唯一标识（秩，rank），从 0 到 size-1。
MPI_Comm_size(comm, &size);
// 获取通信器 comm 中总的进程数。
```

MPI 程序运行时，每个进程是独立的执行单元，运行相同的程序代码。通过 rank，进程可以区分自己的角色（例如，主进程通常是 rank == 0）。size 告诉每个进程参与计算的总进程数，用于任务分配。在本程序中，comm 是通信器（通常为 MPI_COMM_WORLD），定义了参与通信的进程组。

具体来说原因如下：程序需要 rank 来确定每个进程负责的任务（例如，计算点值乘法的子集），同时还需要 size 来计算每个进程的任务分配范围（例如，将点值乘法的任务分成 size 份）。这些信息是并行程序的基础，用于协调进程行为和分配工作。

数据广播 在数据广播方面：

```
// ntt_mpi函数里面的数据广播
MPI_Bcast(a.data(), a.size(), MPI_INT, 0, comm);
// poly_multiply函数里面的数据广播
MPI_Bcast(A.data(), lim, MPI_INT, 0, comm);
MPI_Bcast(B.data(), lim, MPI_INT, 0, comm);
```

在上面的代码里面，MPI_Bcast 将数据从一个进程（根进程，这里是 rank == 0）广播到通信器 comm 中的所有其他进程。在 ntt_mpi 中，输入数组 a 被广播，确保每个进程都有相同的输入数据。在 poly_multiply 中，两个输入多项式 A 和 B 被广播，确保所有进程都有完整的多项式数据。

MPI_Bcast(buffer, count, datatype, root, comm) 将 buffer 中的 count 个 datatype 类型的数据从 root 进程广播到 comm 中的所有进程。这里 datatype 是 MPI_INT，表示数据是整数；root 是 0，表示主进程负责提供数据。**广播是一种集合通信操作，所有进程都会同步，确保每个进程在继续执行之前都收到相同的数据**。从而保证了：

- 数据一致性：NTT 需要对整个输入数组进行变换，每个进程必须有完整的输入数据（A 和 B）才能执行 NTT；
- 简化并行逻辑：在 ntt_mpi 中，每个进程独立执行完整的 NTT 变换，因此需要广播数据以确保所有进程操作相同的数据；
- 初始数据分发：主进程（rank == 0）通常负责初始化输入数据（A 和 B），通过广播分发给其他进程，避免每个进程单独读取或初始化数据。

并行点值 在 poly_multiply 函数中，点值乘法被并行化：

```
int chunk = (lim + size - 1) / size; // 上取整
int start = rank * chunk;
int end = std::min(start + chunk, lim);
std::vector<int> local_C(chunk, 0);
for (int i = start; i < end; ++i) {
    local_C[i - start] = (int)((1LL * A[i] * B[i]) % p);
}
```

上面的并行点值乘法将点值乘法的任务分成 size 份，每个进程计算一部分点值乘法（A[i] * B[i] % p）。chunk 是每个进程需要处理的数据块大小（通过上取整计算）。start 和 end 定义了当前进程（rank）负责的数组索引范围。每个进程将结果存储在本地数组 local_C 中。

点值乘法是多项式乘法中最容易并行化的部分，因为每个点值乘法（A[i] * B[i]）是独立的，互不依赖。通过将数组索引范围 [0, lim) 分成 size 份，每个进程只处理自己的子任务，减少计算时间。任

务分配基于 rank，确保每个进程的工作范围不重叠。

实现上述操作我们可以实现：

- 并行加速：点值乘法是计算密集型操作，平均分配到多个进程可以显著减少总计算时间。
- 负载均衡：通过 chunk 和 start/end 的计算，确保每个进程的工作量大致相等（除了最后一个进程可能处理较少的元素）。
- 局部存储：每个进程使用 local_C 存储自己的计算结果，减少内存需求并避免进程间直接内存访问。

结果收集 在 poly_multiply 函数中我们采取如下的操作：

```
std::vector<int> recv_counts(size);
std::vector<int> displs(size);
for (int i = 0; i < size; ++i) {
    int i_start = i * chunk;
    int i_end = std::min(i_start + chunk, lim);
    recv_counts[i] = i_end - i_start;
    displs[i] = i_start;
}
MPI_Allgatherv(local_C.data(), end - start, MPI_INT, C.data(), recv_counts.data(),
    displs.data(), MPI_INT, comm);
```

使用 MPI_Allgatherv 将每个进程的本地结果 local_C 收集到所有进程的全局数组 C 中。然后继续通过 recv_counts 指定每个进程贡献的数据量，displs 指定每个进程的数据在目标数组 C 中的起始位置。

上述操作的原理在于 MPI_Allgatherv 是一种集合通信操作，允许每个进程发送不同大小的数据块，并将所有数据收集到每个进程的接收缓冲区 (C)。recv_counts[i] 表示进程 i 发送的数据量，displs[i] 表示进程 i 的数据在目标数组 C 中的偏移量。与 MPI_Gather 不同，MPI_Allgatherv 确保所有进程都收到完整的 C 数组，而不仅仅是根进程。

全局结果需求：逆 NTT 需要完整的点值乘法结果（C 数组），因此所有进程都需要收集完整的 C。灵活性：由于最后一个进程可能处理的数据量较少（lim 不一定能被 size 整除），MPI_Allgatherv 支持变长数据收集，适合这种场景。同步性：MPI_Allgatherv 是一个同步操作，确保所有进程在继续执行逆 NTT 之前都拥有相同的 C 数据。

主进程输出结果 在 poly_multiply 函数中：

```
if (rank == 0) {
    for (int i = 0; i < 2 * n - 1; ++i) {
        ab[i] = C[i] % p;
    }
}
```

这里只有主进程（rank == 0）负责将最终结果 C 复制到输出数组 ab 中。MPI 程序通常由主进程负责最终结果的输出或进一步处理。这里使用条件语句 if (rank == 0) 限制只有主进程执行输出操作，避免多个进程同时写入输出数组。这样做能够

- 避免冲突：如果所有进程都试图写入 ab，可能导致数据竞争或重复写入。
- 简化输出：通常只需要一个进程（主进程）将结果返回给调用者或输出到文件/终端。
- 符合 MPI 模式：主进程通常负责协调和最终结果的收集，这是一种常见的 MPI 编程模式。

2.3 算法简介

下面是我们的算法，具体代码可以在我的 GitHub 上面找到：

Algorithm 1 MPI 并行多项式乘法

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

function PolyMultiply(a, b, ab, n, p, comm)
  获取进程秩 `rank` 和进程总数 `size` ▷ `MPI_Comm_rank`, `MPI_Comm_size`
  $\ell \leftarrow 2^{\lceil \log_2(2n-1) \rceil}$ ▷ 扩展到 2 的幂
  初始化数组 $A[\ell], B[\ell], C[\ell] \leftarrow 0$
  if `rank` $= 0$ **then**
    将 $a[0\ldots n-1]$、$b[0\ldots n-1]$ 复制到 A、B，模 p
  end if
  广播 A 和 B 到所有进程 ▷ `MPI_Bcast`
  root $\leftarrow$ GetPrimitiveRoot(p) ▷ 获取原根
  NTT_MPI(A, false, root, p, comm) ▷ 前向 NTT
  NTT_MPI(B, false, root, p, comm)
  chunk $\leftarrow \lceil \ell/\text{size} \rceil$, start $\leftarrow$ rank $\times$ chunk, end $\leftarrow \min(\text{start} + \text{chunk}, \ell)$
  local_C[chunk] $\leftarrow 0$
  for $i =$ start to end $- 1$ **do**
    local_C[$i -$ start] $\leftarrow (A[i] \times B[i]) \bmod p$ ▷ 点值乘法
  end for
  收集 local_C 到 C ▷ `MPI_Allgatherv`
  NTT_MPI(C, true, root, p, comm) ▷ 逆 NTT
  if `rank` $= 0$ **then**
    将 $C[0\ldots 2n-2]$ 复制到 ab，模 p
  end if
end function
function NTT_MPI(a, invert, root, mod, comm)
  获取进程秩 `rank` 和进程总数 `size`
  广播 a 到所有进程 ▷ `MPI_Bcast`
  NTT(a, invert, root, mod) ▷ 串行 NTT
end function

3 实验设计：与多线程以及 SIMD 的结合

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。由于时间关系这里只进行理

论分析和部分代码结构和程序分析，具体实验等内容放到期末大报告之中。

3.1 与 OpenMP 的结合

在我们实现的 `ntt_mpiomp.h` 文件中，**快速数论变换（NTT）通过 OpenMP 实现高效并行化**。位反转置换通过 `bit_reverse` 函数串行执行，确保输入数组 `a` 按正确顺序排列，为后续蝶形运算做准备。蝶形运算在 `ntt` 函数中使用 `#pragma omp parallel` 和 `#pragma omp for schedule(static)` 并行化外层循环，每个线程处理不同的数据块，执行加权和与差计算，并通过模运算确保结果正确，静态调度确保任务均匀分配。逆 NTT 时的归一化操作通过 `#pragma omp for schedule(static)` 并行化，将数组元素乘以模逆 `inv_n`，线程独立处理以避免数据竞争。

在 `poly_multiply` 函数中，MPI 与 OpenMP 结合实现分布式并行计算，根进程初始化并广播输入数组 `A` 和 `B`，各进程调用 `ntt` 执行变换，点值乘法通过分块分配给进程并结合 OpenMP 并行化。然而，过多线程可能导致内存访问冲突，频繁的 `#pragma omp barrier` 同步可能增加开销，需通过优化线程数或减少同步点提升性能。

3.2 与 Pthread 的结合

快速数论变换（NTT）通过结合 Pthread 实现高效并行化。**核心步骤包括位反转置换、蝶形运算和归一化处理**。和上面一样，位反转置换通过 `bit_reverse` 函数利用 Pthread 将输入数组 `a` 分块分配给多个线程，每个线程独立处理索引范围，通过 `pthread_create` 和 `pthread_join` 管理线程以避免数据竞争。蝶形运算在 `ntt` 函数中通过 `ntt_butterfly_thread` 并行执行，线程处理不同数据块的加权和与差计算，使用 `safe_mod` 确保模运算安全。逆 NTT 时的归一化操作则通过 `ntt_normalize_thread` 并行化，将数组元素乘以模逆 `inv_n`，任务按元素分块以避免冲突。

在 `poly_multiply` 函数中，MPI 与 Pthread 结合实现分布式并行计算，根进程初始化并广播输入数组 `A` 和 `B`，各进程调用 `ntt` 执行变换，点值乘法通过 Pthread 并行化处理。然而，Pthread 的线程创建和同步开销在小规模数据上可能较为显著，且多线程访问共享数组可能导致缓存失效，需优化线程数和数据局部性以提升性能。

3.3 与 SIMD 的结合

在我们实现的 `ntt_mpisimd.h` 文件中，快速数论变换（NTT）通过 ARM NEON SIMD 和 MPI 实现高效并行化。核心步骤包括位反转置换、蝶形运算和归一化处理。位反转置换通过串行方式在 `ntt_simd` 函数中完成，确保输入数组 `a` 按正确顺序排列，为后续运算准备数据。蝶形运算利用 `butterfly_layer_simd` 函数，通过 NEON 指令并行处理四组数据的加权和与差计算，借助 `mod_reduce_neon` 优化模运算，显著提升计算效率。归一化阶段在逆 NTT 时通过 SIMD 向量化处理，将数组元素乘以模逆 `inv_n`，并对剩余元素进行标量处理以确保正确性。在 `poly_multiply` 函数中，MPI 广播输入数组 `A` 和 `B`，各进程执行 NTT 变换，点值乘法通过 SIMD 并行化并分块分配，根进程完成逆 NTT 并输出结果。

然而，SIMD 实现的性能可能因多种因素而变慢。**NEON 指令的向量化要求数据对齐和连续内存访问，若输入规模较小或数据未正确对齐，会导致缓存未命中率升高，降低效率**。此外，`butterfly_layer_simd` 中频繁的内存分配和释放（如 `twiddles` 数组）增加开销，尤其在高频调用时。MPI 通信（如 `MPI_Bcast` 和 `MPI_Allgatherv`）在大数据量下可能引入延迟，尤其是当进程数较多时，通信开销显著。相比 OpenMP 的并行循环，SIMD 的复杂指令序列和内存管理可能导致性能瓶颈，特别是在测试中遇到部分数据无法运行的情况，可能是由于模数或数组大小的边界检查不足，需优化内存管理和通信策略以提升稳定性与速度。

4 实验设计：MPI 并行化不同算法策略

4.1 巴雷特规约优化

我们在之前的实验中其实初步尝试了巴雷特规约这种方法，接下来我们继续在这一部分进行解释：

巴雷特规约（Barrett Reduction）是一种高效的模运算优化算法，用于计算 $x \mod p$，特别适用于模数 p 固定且需要多次模运算的场景（如快速数论变换 NTT）。通过预计算和位运算，它将昂贵的除法操作替换为乘法和减法，从而提升性能。

原理 巴雷特规约的目标是高效计算 $x \mod p$，其中 x 是非负整数（通常 $x < p^2$），p 是模数。传统模运算需要计算：

$$x \mod p = x - \left\lfloor \frac{x}{p} \right\rfloor \cdot p$$

其中，商 $q = \lfloor x/p \rfloor$ 的计算涉及昂贵的除法操作。巴雷特规约通过预计算倒数近似值，将除法转换为乘法和位运算。

设 k 为满足 $2^k > p$ 的整数，通常取 $k \approx 2 \cdot \lceil \log_2 p \rceil$。预计算常数：

$$m = \left\lfloor \frac{2^k}{p} \right\rfloor$$

商 q 可近似为：

$$q \approx \left\lfloor \frac{x \cdot m}{2^k} \right\rfloor$$

在计算机中，这可以通过位运算实现：

$$q = (x \cdot m) \gg k$$

最终模结果为：

$$r = x - q \cdot p$$

由于 q 是近似值，r 可能比实际模大 p 或 $2p$，因此需检查 $r \geq p$ 并进行 1-2 次减法以确保 $0 \leq r < p$。

步骤

1. **预计算**：选择 k，计算 $m = \lfloor 2^k/p \rfloor$。
2. **计算商**：对于输入 x，计算 $q = (x \cdot m) \gg k$。
3. **计算模**：计算 $r = x - q \cdot p$，若 $r \geq p$，重复减去 p。
4. **返回结果**：返回 r 作为 $x \mod p$。

优化效果 巴雷特规约将除法替换为一次乘法、一次位移和少量减法。在现代处理器上，乘法和位移的开销远低于除法（除法约几十周期，乘法约 1-3 周期）。对于 NTT 等模运算密集的算法，性能可提升 5-10 倍。预计算 m 仅需一次，适合固定模数的场景。

注意事项

- **选择** k：通常 $k = 2 \cdot \lceil \log_2 p \rceil$，确保 $x < p^2$。

- **溢出**：乘法 $x \cdot m$ 可能溢出，需使用 128 位整数（如 C++ 的 `__int128`）。
- **适用范围**：最适合 $x < p^2$，如 NTT 中的点值乘法。

伪代码 以下是巴雷特规约的伪代码：

Algorithm 2 巴雷特规约

Input: 整数 x，模数 p，预计算 $m = \lfloor 2^k/p \rfloor$，$k \approx 2 \cdot \lceil \log_2 p \rceil$

Output: $r = x \mod p$

```
function BarrettReduction(x, p, m, k)
    q ← (x · m) ≫ k                                  ▷ 位运算近似商
    r ← x − q · p
    while r ≥ p do
        r ← r − p
    end while
    return r
end function
```

具体到我们的程序而言是这样的：

Algorithm 3 巴雷特规约算法

Input: 输入值 x，模数 p（p 为正整数，通常为 NTT 模数，如 7340033 或 998244353）

Output: 规约结果 $r \equiv x \pmod p$，且 $0 \le r < p$

```
function BarrettReduction(x, p)
    预计算：k ← ⌊log2 p⌋ + 1                          ▷ 计算模数的位长度
    预计算：m ← ⌊2^(2k)/p⌋                            ▷ 预计算巴雷特常数
    q ← ⌊(x · m)/2^(2k)⌋                              ▷ 估算商
    r ← x − q · p                                     ▷ 计算余数
    while r ≥ p do                                    ▷ 确保结果在 [0, p) 范围内
        r ← r − p
    end while
    while r < 0 do                                    ▷ 处理负数情况
        r ← r + p
    end while
    return r
end function
```

4.2 常规的四分优化

这里我们仿照上次的 Pthread 优化操作操作就可以了，把 pthread 里实现的 dif/dit 和 crt 合并在多进程里再实现一次即可，当然可以尝试将 crt 合并的模数分组计算，同一进程内使用多个线程进行计算，最后多个进程再统一合并。需要注意的是，如果实现了四分 NTT，需要对于奇数进行额外一次的变换才能继续四分。（此处代码视时间情况而定，报告相关内容后续会放到大报告当中）

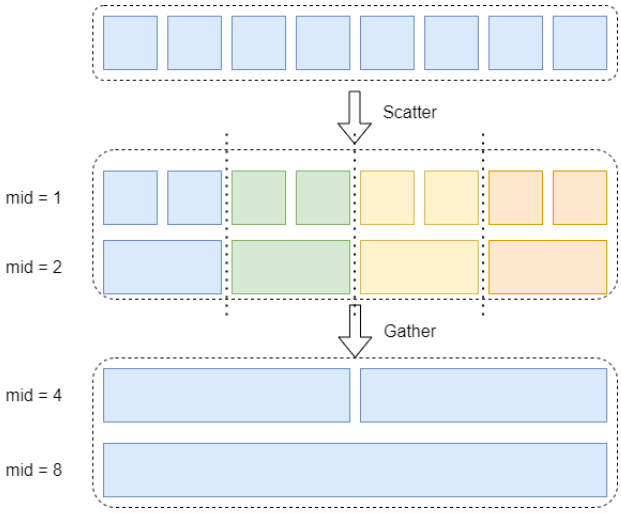

图 4.1: 多分的一个示例，来自实验指导手册

5 实验设计：MPI 不同的编程方法

由于 MPI 特性的加持，所以我们除了对算法的优化以外，我们继续讨论一下多种 MPI 上的编程方法对于这次实验的影响。

5.1 阻塞通信 VS 非阻塞通信

不难发现，我们刚才的实验中使用了阻塞通信，这在某种程度上会影响我们程序的性能。

阻塞通信是消息传递接口（MPI）编程模型中的核心通信方式，深刻影响进程间数据交互的逻辑与系统资源利用效率。在阻塞通信中，进程调用通信函数后会暂停执行，进入阻塞状态，直到通信操作完成才能继续运行。这种机制确保数据传输的可靠性，但可能导致资源利用效率降低。

对于非阻塞通信，不必等到通信操作完全完成便可以返回，该通信操作可以交给特定的通信硬件去完成，在该通信硬件完成该通信操作的同时，处理机可以同时进行计算操作，这样便实现了计算与通信的重叠。

阻塞通信的机制 阻塞通信包括以下两种核心操作：

- **阻塞式发送**：进程调用发送函数（如 `MPI_Send`）后，系统开始传输数据。进程会暂停，等待数据完全送入缓冲区并可被接收方访问。系统需完成数据打包、网络传输准备和缓冲区写入等步骤，确认消息可靠发送后，函数返回，进程继续执行。
- **阻塞式接收**：进程调用接收函数（如 `MPI_Recv`）后进入等待状态，监听目标消息的到来，无法执行其他计算或通信任务。只有当完整消息抵达接收缓冲区并通过校验，函数才会返回。例如，在并行矩阵乘法中，进程可能需等待其他进程的中间结果以完成最终计算。

特点与影响 阻塞通信保证数据顺序与可靠性，适合矩阵分解等严格依赖场景。但等待期间处理器闲置，资源利用率下降，尤其在高性能集群中可能延长执行时间。与非阻塞通信（如 `MPI_Isend/MPI_Irecv`）相比，阻塞通信无法重叠计算与通信，逻辑简单但效率较低，适合初学者或同步要求高的场景。其原理也是类似于计算机组成原理中的五级流水线操作。

5.2 单边通信 VS 多边通信

单边通信（One-Sided Communication） 单边通信指单个进程主动发起对远程进程内存的访问操作，无需目标进程配合执行接收操作的通信模式，又称远程内存访问（RMA，Remote Memory Access）。**核心特点**如下：

- **主动访问机制**：发起进程可直接读写目标进程的内存，无需目标进程显式调用接收函数。
- **典型操作**：
  - `Put`：将数据从本地内存复制到远程进程内存。
  - `Get`：从远程进程内存读取数据到本地。
  - `Accumulate`：对远程数据执行原子操作（如加法、最大值更新等）。
- **同步需求**：通过同步原语（如 `MPI_Fence`、`MPI_Lock`、`MPI_Unlock`）确保数据一致性，避免访问冲突。
- **应用场景**：适用于稀疏矩阵运算、非结构化网格通信、异步数据交换等场景，减少进程间同步等待。

示例场景：在并行计算中，某进程需频繁读取多个远程进程的局部数据，无需等待对方响应时，可使用单边通信提升效率。

但是在实际过程中，MPI 单边通信中的内存重叠问题导致的。具体来说当使用 MPI_Put 将数据放入自己的进程窗口时，源地址和目标地址相同，这会导致 MPI 内部 memcpy 操作失败。这可能与 MPI 单边通信中的内存管理或同步问题有关。代码中 MPI_Win_create 和 MPI_Put/MPI_Get 操作可能导致窗口缓冲区与本地缓冲区重叠，尤其是在处理大数组（padded_lim 262144）时。MPI 实现的内部缓冲区管理或同步机制也可能是问题来源。通过使用独立缓冲区、优化同步、限制数组大小和调试内存地址，可以有效定位和解决问题。

多边通信（Multi-Sided Communication） 多边通信指多个进程同时参与的数据交互模式，包括集体通信（Collective Communication）和多方点对点通信。**核心类型与特点**如下：

1. **集体通信**
   - **定义**：所有参与进程共同执行的协同操作，需统一调用相同 API。
   - **常见操作**：
     - `Broadcast`：根进程向所有进程发送相同数据。
     - `Reduce`：将各进程数据聚合（如求和、最大值）到根进程。
     - `Scatter`：根进程将数据分发给所有进程。
     - `Gather`：各进程数据汇总到根进程。
   - **特点**：需所有进程同步参与，通信逻辑统一，适合数据并行场景。
2. **多方点对点通信**
   - **定义**：多个进程通过多次点对点通信（如 `Send/Recv`）实现数据交互。
   - **特点**：灵活性高，但需手动管理通信顺序，避免死锁。

应用场景：集体通信适用于数据初始化（如广播）、结果汇总（如归约）；多方点对点通信适用于多节点任务调度、动态负载均衡等场景。

表 2: 单边通信与多边通信（集体通信）对比

维度	单边通信	多边通信（集体通信）
通信发起方	单个进程主动访问远程内存	所有进程协同参与，需同步调用 API
目标进程配合	无需目标进程显式接收	所有进程必须执行相同操作
同步机制	依赖 RMA 同步原语（如 `Fence/Lock`）	依赖集体操作的隐式同步
典型操作	`Put/Get/Accumulate`	`Broadcast/Reduce/Scatter/Gather`
适用场景	稀疏通信、异步数据访问	数据并行计算、全局数据聚合

单边通信与多边通信的对比 单边通信通过 RMA 机制实现单进程对远程内存的主动访问，适合减少同步开销的场景；多边通信（尤其是集体通信）强调多方协同，适合全局数据交互。两者可根据任务的通信模式结合使用，以优化性能和编程效率。

5.3 MPI 自身的多线程支持

MPI（Message Passing Interface）的多线程支持允许程序在单个进程内结合多线程并行与跨进程通信，适用于异构计算环境（如多核 CPU+GPU）或需要细粒度任务调度的场景。

MPI 标准定义了 4 种线程支持级别：

- MPI_THREAD_SINGLE：仅单线程可调用 MPI，其他线程调用会导致未定义行为。
- MPI_THREAD_FUNNELED：仅主线程可调用 MPI，其他线程只能执行非 MPI 操作。
- MPI_THREAD_SERIALIZED：多线程可调用 MPI，但同一时刻仅单一线程能执行 MPI 函数（需用户自行保证序列化）。
- MPI_THREAD_MULTIPLE：完全支持多线程并行调用 MPI 函数（需 MPI 实现底层线程安全）。

结合多线程（共享内存）与 MPI（分布式内存），减少进程创建开销，优化数据局部性（如线程内缓存复用）。但是实际应用中，我们需处理线程安全（如 MPI 函数重入性）、负载均衡（避免线程饥饿）及不同 MPI 实现的兼容性（如 Open MPI、MPICH 对多线程的支持差异）。

在后面的实验中，我们会继续使用相应的接口，所以这一部分暂时不会列出具体的代码，因为我们在前面的实验中已经充分利用了这一点。

6 程序性能分析

这次实验，我们采用下面的编译方式：

```
mpic++ main.cc -o main -O2 -fopenmp -lpthread -std=c++11
qsub qsub_mpi.sh
```

这是助教在课上提到的编译方式，我们先编译再提交，提交脚本里面可以指定-np 的数量，决定 MPI 线程数。

```
/usr/local/bin/mpiexec -np 8 -machinefile $PBS_NODEFILE /home/${USER}/main
```

配置	n	p	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
MPI (8 线程)	4	7340033	0.1612	0.0264
	131072	7340033	110.8640	0.7572
	131072	104857601	99.2098	0.8460
	131072	469762049	97.7614	0.8588
MPI (OpenMP, 8 线程)	4	7340033	84.2481	0.0001
	131072	7340033	268.8230	0.3123
	131072	104857601	251.9930	0.3331
	131072	469762049	251.6210	0.3336
MPI (pthread, 8 线程, 8 pthread 线程)	4	7340033	7.6301	0.0006
	131072	7340033	209.5740	0.4005
	131072	104857601	226.8320	0.3701
	131072	469762049	230.1110	0.3648

表 3: 8 线程配置及 NTT 实现的多项式乘法平均延迟与加速比

6.1 不同并行化方式的对比

由于随机性和相差不大的原因，这里不考虑小规模数据（n=4），如表3所示，进行优化之后的算法当规模增大的时候，

- **MPI（8 线程）**：
  - 在小规模输入（$n = 4, p = 7340033$）下，平均延迟为 0.1612 μs，性能较基准下降，但仍远高于大规模输入的延迟。
  - 在大规模输入（$n = 131072$）下，平均延迟在 97.7614 μs 至 110.8640 μs 之间，低于基准（加速比 0.7572–0.8588），表明 MPI 在大规模计算中性能略逊于 NTT 实现，模数增大对延迟影响较小。
- **MPI（OpenMP，8 线程）**：
  - 在小规模输入（$n = 4, p = 7340033$）下，平均延迟为 84.2481 μs，性能显著低于基准，可能是 OpenMP 引入的线程开销。
  - 在大规模输入（$n = 131072$）下，平均延迟在 251.6210 μs 至 268.8230 μs 之间，远高于基准（加速比 0.3123–0.3336），表明 OpenMP 在大规模计算中性能较差，模数变化对延迟影响有限。
- **MPI（pthread，8 线程，8 pthread 线程）**：
  - 在小规模输入（$n = 4, p = 7340033$）下，平均延迟为 7.6301 μs，性能较基准大幅下降，可能由于 pthread 线程管理开销。
  - 在大规模输入（$n = 131072$）下，平均延迟在 209.5740 μs 至 230.1110 μs 之间，远高于基准（加速比 0.3648–0.4005），表明 pthread 配置在大规模计算中性能较差，模数增大导致延迟略增。

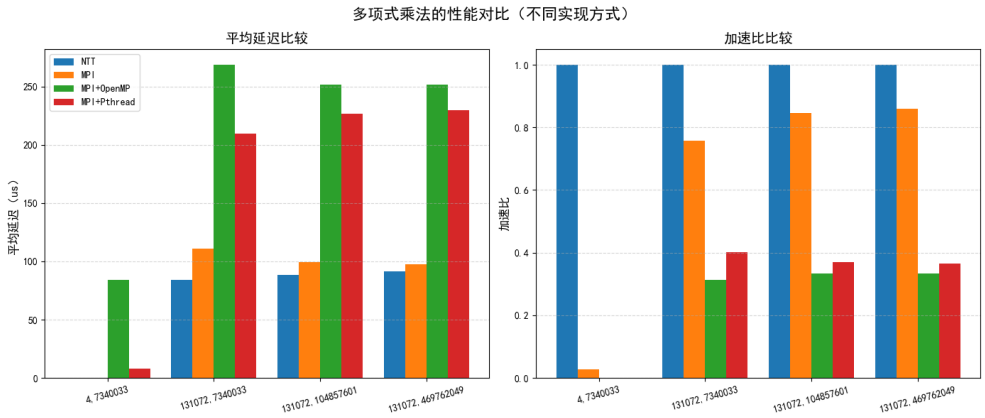

图 6.2: 延迟时间和加速比对比

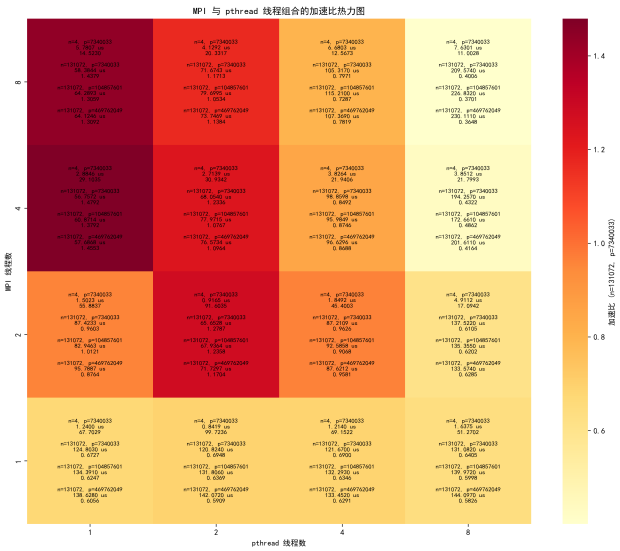

图 6.3: 加速比可视化，基于 MPI 与 Pthread 的结合

6.2 对线程数的讨论

通过上面的数据可以看到，开了 8 线程之后，加速效果反而不是那么明显了，接下来我们要开始讨论线程数对实验的影响：

这张图对应的表格放在附录的表7。可以看到图6.3，mpi 的线程数和 pthread 的线程数并不都是越高越能加速实验运行时间，而是要达到一个“平衡”。在 pthread 线程数设置为 1 的时候，**mpi 的线程数越大，越有助于加速计算**。证明二者是相辅相成的。

MPI 线程数过大可能会导致资源过度消耗、内存不足、线程切换和通信开销增加等问题，进而引发性能下降、负载不均衡、调试难度加大以及死锁风险，最终影响程序的可扩展性和整体效率。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——也就是后面对 OpenMP 的讨论。

6.3 不同规约优化方法下的对比

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。

配置	n	p	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
MPI (巴雷特, 8 线程)	4	7340033	0.1168	0.0364
	131072	7340033	95.4719	0.8794
	131072	104857601	97.2662	0.8630
	131072	469762049	100.0120	0.8395
MPI (提升版, 8 线程)	4	7340033	0.0639	0.0665
	131072	7340033	82.0365	**1.0234**
	131072	104857601	78.7015	**1.0667**
	131072	469762049	77.8449	**1.0784**

表 4: 8 线程配置及 NTT 实现的多项式乘法平均延迟与加速比

我们能够看到表4所示的数据：

- **MPI（巴雷特，8 线程）**：
  - 在小规模输入（$n = 4, p = 7340033$）下，平均延迟为 0.1168 μs，性能优于标准 MPI，但仍低于 NTT 基准。
  - 在大规模输入（$n = 131072$）下，平均延迟在 95.4719 μs 至 100.0120 μs 之间，接近基准（加速比 0.8395–0.8794），显示巴雷特优化在高模数下略有提升，但整体性能与 NTT 相近。
- **MPI（提升版，8 线程）**：
  - 在小规模输入（$n = 4, p = 7340033$）下，平均延迟为 0.0639 μs，性能优于其他 MPI 配置，但仍低于 NTT 基准。
  - 在大规模输入（$n = 131072$）下，平均延迟在 77.8449 μs 至 82.0365 μs 之间，**优于基准（加速比 1.0234–1.0784）**，尤其在高模数（$p = 469762049$）下表现最佳（加速比 1.0784），显示提升版优化在大规模计算中效果显著。

6.4 对 OpenMP 的讨论

由于 OpenMP 的接触表现，接下来我们将它与 MPI 结合，看看有什么精彩的表现：

如图6.4所示，不难发现在 OpenMP 的作用下 MPI 线程数越小，程序加速越明显：二者结合发挥的威力更大。

- **MPI 结合 OpenMP（8 线程）**：
  - 延迟 251.621 μs–268.823 μs，加速比 0.3123–0.3336，性能远低于基准，p=469762049 略优（0.3336）。
- **MPI 结合 OpenMP（4 线程）**：
  - 延迟 141.426 μs–142.573 μs，加速比 0.5889–0.5936，优于 8 线程但低于基准，p=469762049 稍优（0.5936）。

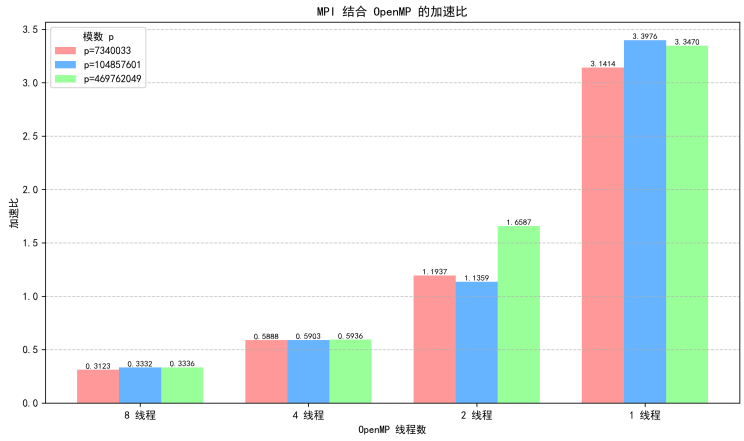

图 6.4: 加速比可视化

- **MPI 结合 OpenMP（2 线程）**:
  - 延迟 50.6141 ţs–73.9069 ţs，加速比 1.1358–1.6587，优于基准，p=469762049 最佳（1.6587）。
- **MPI 结合 OpenMP（1 线程）**:
  - 延迟 24.7095 ţs–26.7246 ţs，加速比 3.1413–3.3980，远超基准，p=104857601 最佳（3.3980）。

究其原因还是因为 MPI 之间开了较大的线程数会导致之间的通信出现延迟甚至阻塞，进而引发计算效率的问题。有的时候甚至会导致计算错误的问题。

一些可能的解决方法：测试 MPI 线程安全级别（使用 MPI_Init_thread）优先选择 MPI_THREAD_FUNNELED 或 MPI_THREAD_SERIALIZED，避免随意使用 MPI_THREAD_MULTIPLE。还可以设置线程数接近物理核心数（如每个节点 1 个 MPI 进程 + 每个进程 16 线程），并通过基准测试（如弱缩放测试）调优。使用工具（如 mpiP、Vampir）监控通信开销和线程行为。总而言之，为了避免问题，应当限制每个进程的线程数，**确保通信操作由少数线程管理；采用异步通信减少阻塞**。

7 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 -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，只考虑并行化操作之间的优势）。

7.1 数据的总览

需要注意的是这里的各个优化都是基于 MPI 完成的，而不是单独使用了某一种方法。

指标	mpi 普通	pthread	巴雷特规约	mpi 提升	OpenMP
CPU 时钟周期 ($\times 10^9$)	15.266	5.489	22.201	5.661	**19.163**
指令数量 ($\times 10^9$)	2.299	2.128	2.380	2.417	**5.406**
每周期指令数	1.446	1.998	1.419	1.498	**2.258**
每指令缓存引用	0.295	0.287	0.297	0.295	**0.218**
缓存未命中率 (%)	0.809	0.504	0.858	0.951	**0.327**
L1 数据缓存未命中率 (%)	0.809	0.503	0.859	0.950	**0.328**
最后级缓存未命中率 (%)	18.673	16.315	14.378	16.757	**23.241**

表 5: 优化方法间的硬件性能对比分析（NTT 朴素只作为 baseline，不参与比较）

指标	2__1	2__2	2__4	2__8
CPU 时钟周期 ($\times 10^9$)	**2.100**	5.448	11.438	78.915
指令数量 ($\times 10^9$)	**1.603**	3.844	8.480	15.431
每周期指令数	**2.573**	1.765	1.573	1.124
每指令缓存引用	**0.338**	0.397	0.424	0.420
缓存未命中率 (%)	0.135	0.124	**0.096**	0.239
L1 数据缓存未命中率 (%)	0.136	0.122	**0.096**	0.235
最后级缓存未命中率 (%)	46.426	**25.582**	44.983	40.274

表 6: 不同线程配置下的硬件性能对比分析，第一个数字是 num_thread 的数值，第二个数字是 mpi 的线程数

7.2 进一步的解释

第一张表 分析如下：

由表5可以知道，对于需要高效利用 CPU 和缓存资源的场景，**OpenMP 和 MPI 结合优化是最佳选择**。其在每周期指令数（2.258）、缓存未命中率（0.327%）和 L1 数据缓存未命中率（0.328%）上表现优异，展现了良好的指令级并行性和缓存利用效率。然而，OpenMP 的 CPU 时钟周期（19.163 $\times 10^9$）和最后级缓存未命中率（23.241%）较高，可能是线程同步开销或内存访问模式未完全优化导致。

pthread 优化在 CPU 时钟周期（5.489 $\times 10^9$）上表现最佳，接近 mpi 提升优化，但在缓存未命中率（0.504%）和最后级缓存未命中率（16.315%）上表现一般。可以通过优化 `ntt_layer_thread` 函数，减少线程同步开销，并改进内存访问模式以进一步降低缓存未命中率。

巴雷特规约优化和 **mpi 提升优化**在性能上较为接近，但 CPU 时钟周期（巴雷特规约: 22.201 $\times 10^9$，mpi 提升: 5.661 $\times 10^9$）和指令数量（巴雷特规约: 2.380 $\times 10^9$，mpi 提升: 2.417 $\times 10^9$）较高，表明存在优化空间。可以通过优化 `poly_multiply_single_mod` 和 `__modti3` 函数，减少指令执行数量，并改进数据局部性以提升缓存命中率。

所有优化方法均可通过改进内存访问模式进一步降低缓存未命中率，特别是 **pthread 优化**和 **OpenMP 优化**，其最后级缓存未命中率（分别为 16.315% 和 23.241%）相对较高。未来可重点优化内存分配和数据预取策略，以提升整体性能。

第二张表 基于前面 MPI+Pthread 谈谈 Pthread 线程数一定，MPI 线程数对硬件性能的影响：

由表6可以知道，对于需要高效利用 CPU 和缓存资源的场景，**2__1 配置（num__thread=2, mpi=1）**在 CPU 时钟周期（2.100 $\times 10^9$）和指令数量（1.603 $\times 10^9$）上表现最佳，适合低开销场景。然而，其最后级缓存未命中率（46.426%）较高，表明内存访问模式有优化空间。相比之下，**2__4 配置（num__thread=2, mpi=4）**在缓存未命中率（0.096%）和 L1 数据缓存未命中率（0.096%）上表现最佳，适合需要高效

缓存利用的场景。

2__2 配置在最后级缓存未命中率(25.582%)上表现最佳,但 CPU 时钟周期(5.448×10^9)和指令数量(3.844×10^9)较高,可能因线程同步或数据局部性问题导致性能瓶颈。可以通过优化 `ntt_layer_thread` 函数，减少线程间通信开销，并改进内存访问模式以进一步提升性能。

2__4 配置和 **2__8 配置**在指令数量（分别为 8.480×10^9 和 15.431×10^9）和 CPU 时钟周期（分别为 11.438×10^9 和 78.915×10^9）上较高，表明并行度增加导致了额外的计算和同步开销。可以通过优化 `poly_multiply_single_mod` 和 `__modti3` 函数，减少指令冗余，并改进数据分区以提升缓存命中率。

所有配置均可通过改进内存访问模式降低缓存未命中率，特别是 **2__1 配置**和 **2__8 配置**，其最后级缓存未命中率（分别为 46.426% 和 40.274%）较高。未来可重点优化数据预取和内存分配策略，以提升整体性能。

7.3 数据分析

第一张表 对不同规约化和并行化操作的总结:

1. **OpenMP 优化**在大多数性能指标上表现最佳，特别是在每周期指令数（2.258）、缓存未命中率（0.327%）和 L1 数据缓存未命中率（0.328%）上，展现了高效的指令级并行性和优异的缓存利用率。然而，其 CPU 时钟周期（19.163×10^9）和最后级缓存未命中率（23.241%）较高，可能受限于线程同步开销或内存访问模式。

2. **pthread 优化**在 CPU 时钟周期（5.489×10^9）上表现突出，仅次于 mpi 提升优化，但缓存未命中率（0.504%）和最后级缓存未命中率（16.315%）偏高。其主要瓶颈可能在于 `ntt_layer_thread` 函数的线程管理开销和内存访问效率。

3. **巴雷特规约优化**和 **mpi 提升优化**在指令数量（分别为 2.380×10^9 和 2.417×10^9）和 CPU 时钟周期（分别为 22.201×10^9 和 5.661×10^9）上表现相近，均存在较大的优化空间。主要瓶颈可能在 `poly_multiply_single_mod` 和 `__modti3` 函数，需减少指令冗余并优化数据局部性。

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 > pthread > mpi 提升 > 巴雷特规约**。

这里主要是因为 MPI 通信开销吞噬计算收益，有效计算效率低。根源在于进程间通信延迟：MPI 进程需通过网络交换数据（如 MPI_Send/Recv），而 pthread/OpenMP 仅需共享内存通信，延迟差可达 100 倍。同步阻塞：集体操作强制所有进程等待最慢者，而其高时钟周期反映负载不均衡导致的空闲等待。内存访问模式劣化问题源于数据分散性：MPI 进程独占内存空间，计算需显式数据分发（如 MPI_Scatter），破坏空间局部性。跨节点访问方面若进程跨 NUMA 节点，访问远端内存进一步增加延迟（对比 OpenMP 的 0.327% L1 未命中率）。

最后还有并行粒度与负载失衡现象：MPI 提升在指令数接近巴雷特规约时，周期却仅为后者的 1/4，说明并行化有效但开销大。源于粗粒度任务划分：MPI 通常以进程为单位分割大任务，若任务粒度不均（如 `poly_multiply_single_mod` 计算量波动），导致部分进程早完工等待（时钟周期虚高）。

总的来说有三：**通信延迟、内存隔离、负载失衡**。也印证了 OpenMP+MPI 的有效性。

第二张表 接下来我们继续分析不同 mpi 的线程的影响:

1. **2__1 配置**在 CPU 时钟周期（2.100×10^9）、指令数量（1.603×10^9）和每周期指令数（2.573）上表现最佳，适合计算资源受限的场景。而其最后级缓存未命中率（46.426%）最高，可能受限于内存访问模式。

2. **2_2 配置**在最后级缓存未命中率（25.582%）上表现优异，但 CPU 时钟周期和指令数量较高，可能因线程同步开销或数据局部性不足导致，需优化 ntt_layer_thread 函数。

3. **2_4 配置**在缓存未命中率(0.096%)和 L1 数据缓存未命中率(0.096%)上表现最佳,适合需要高效缓存利用的场景,但高 CPU 时钟周期(11.438 $\times 10^9$)表明存在并行开销,需优化 poly_multiply_-single_mod 和 ___modti3 函数。

4. **2_8 配置**在所有指标上表现较差，尤其是 CPU 时钟周期（78.915 $\times 10^9$）和指令数量（15.431 $\times 10^9$），表明高并行度带来了显著的同步和计算开销，需大幅优化内存访问和线程管理。

5. 从指令执行效率和缓存利用率来看，四种配置的排名为：**2_1 > 2_2 > 2_4 > 2_8**。

总结来说，MPI 性能问题的本质是资源竞争与访问劣化的正反馈循环：**线程数增加 通信请求激增 网络/NIC/缓冲区竞争 → 线程阻塞等待 → 内存访问分散 → 缓存失效 → 更多通信需求 → 性能崩溃**。表6中 2_8 的劣化正是这一链条的终极体现，而 2_1 的高效印证了粗粒度并行在 MPI 中的普适性。需通过**限制每进程线程数、解耦通信与计算、通信模式优化、内存访问重构优化**打破此循环。

8 实验总结

实验的代码在我的 GitHub 网站上：我的 GitHub

在使用 MPI 进行算法优化时，我尝试了很多方法试着优化，但是结果往往没有什么改进甚至是负优化。究其原因，是因为再调用 MPI 线程的时候，可能会存在一些阻塞甚至冲突的问题，在这个过程中我逐渐注意这一点。当然过程中也遇到了一些我无法解决的问题，例如使用单边通信或者非阻塞方式都不能一一解决，由于时间关系我不得不放在最后的大报告中进行陈述。

这次实验在考试周袭击而来的 6 月，我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。

附录 A 实验的表格

MPI 进程数	Pthread 线程数	$p = 7340033$	$p = 104857601$	$p = 469762049$
8	8	209.574	226.832	230.111
8	4	105.317	115.210	107.369
8	2	71.674	79.700	73.747
8	1	58.384	64.289	64.125
4	8	194.257	172.661	201.611
4	4	98.860	95.985	96.630
4	2	68.054	77.972	76.573
4	1	56.757	60.871	57.687
2	8	137.522	135.355	133.574
2	4	87.211	92.586	87.621
2	2	65.653	67.936	71.730
2	1	87.423	82.946	95.789
1	8	131.082	139.972	144.097
1	4	121.670	132.293	133.452
1	2	120.824	131.806	142.072
1	1	124.803	134.391	138.628

表 7: MPI 与 Pthread 结合的多项式乘法延迟（n=131072，单位：微秒）