MPI 编程：基于 CRT 的 NTT 优化

2313911 黄尚扬

2025.6.12

目录

一．引言

MPI（Message Passing Interface）是一种广泛应用于分布式内存系统中的并行编程模型，适用于多节点集群环境下的进程间通信与协作。与基于共享内存的 pthread 或 OpenMP 不同，MPI 通过显式的消息传递机制，实现不同计算节点间的数据交换与任务协同，具备良好的可扩展性和高效的跨节点通信能力，常用于大规模科学计算、分布式仿真、图计算等领域。

多项式乘法在密码学中的应用尤为突出，特别是在同态加密、同态签名等需要在密文域中进行高效代数运算的场景中。数论变换（NTT）作为实现多项式快速乘法的关键技术，其在模意义下的性质使其特别适合整数域下的加速运算。而随着加密规模的扩大，传统的串行 NTT 已难以满足高性能计算的需求。

本课题聚焦于 MPI 编程在 NTT 计算中的应用，尝试了基于中国剩余定理（CRT）的 NTT 并行优化方案。通过将大模数乘法问题分解为多个小模下的 NTT 计算，结合 MPI 将任务分发至多个计算节点并行执行，最终使用 CRT 合并结果，不仅减轻了单节点的计算负担，还显著提升了整体吞吐性能。

二．问题描述

由于笔者早已在上次实验（Pthread&OpenMP）中实现了四模数 CRT 的合并，关于 CRT 优化 NTT 的原理我们在这里就不做展开了。下面主要聚焦于 MPI 本身进行思考。

首先我们需要更改原先的计时策略，其次我们需要在合并时阻塞进程来使其不会出现冲突冒险的情况。当我们做到这些之后，与多线程编程相似地，我们就完成了本次的任务。

这次报告中，主要分为实验、性能分析、理论和扩展研究三个部分。

我们的实验主要分为两块：第一是用 MPI 进行多线程朴素的替代；而第二是采用 MPI 结合 OpenMP 进行多进程+多线程的混合优化。至于 SIMD 和其他的 MPI 策略，由于分析发现并没有太大效果，我们主要着重在最后一部分进行分析。

在性能分析部分，我们将会进行大量、不同规模的重复实验，直观地对比并结合代码特征进行评估。

最后一部分，我们会详细解释为什么笔者认为 SIMD 不行，以及为什么数据并行的 MPI 策略似乎是最优的。同时，我们还会对其他 MPI 策略进行一定的探讨。

接下来，我们将从算法逐步展开。

三．算法设计

首先，为了测试 MPI 框架并且对比分析性能，我们会先简单地利用 MPI 代替多线程加速 4 模数 CRT 下的 NTT。

首先，我们会为 MPI 的实现搭建基础框架，包括：

```
 MPI_Init(&argc,&argv);
int rank;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

上面这部分写在 main 函数开头，用于初始化 MPI 环境并为进程 rank 赋值。自然而然地，在结尾处我们同样有：

```
MPI_Finalize();
```

而后由于 MPI 编程的计时方式不同，我们在每轮循环执行主体代码前会有：

```
 auto Start=MPI_Wtime();//MPI 计时
```

并且在**主进程 rank0** 最末尾相似地计时 End，仿照之前的格式做差并输出。特别需要注意的是由于默认计时单位不同，所以需要乘一个 10^3 的常数。这块见后面核心代码的末尾。

而我们的核心代码如下：

```
//下面是多模数 ntt
    for(int J=0;J<=lim;++J){
        ac[rank][J]=a[J]%mod[rank];
        bc[rank][J]=b[J]%mod[rank];
    }
    ntt(ac[rank],bc[rank],ac[rank],n_,mod[rank]);
    MPI_Barrier(MPI_COMM_WORLD);//等待所有进程计算完毕


    if(rank!=0){
```

```
        MPI_Send(ac[rank],lim,MPI_LONG_LONG_INT,0,114514,MPI_COMM_WORLD);
    }
    else{
        for(int j=1;j<4;++j)
            MPI_Recv(ac[j],lim,MPI_LONG_LONG_INT,j,114514,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    //分别执行子任务(MPI)，然后传回主进程

    if(rank==0){//主进程
        __int128 M1M2=mod[0]*mod[1];
        __int128 M1M2M3=M1M2*mod[2];
        __int128 invM1_modM2=ksm(mod[0], mod[1]-2, mod[1]);
        __int128 invM1M2_modM3=ksm(M1M2%mod[2], mod[2]-2, mod[2]);
        __int128 invM1M2M3_modM4=ksm(M1M2M3%mod[3], mod[3]-2, mod[3]);
        for (int j=0; j <= lim; ++j) {
            __int128 x1=((ac[1][j]-ac[0][j])%mod[1]+mod[1])%mod[1];
            x1=x1*invM1_modM2%mod[1];
            __int128 r1=(x1*mod[0]%M1M2+ac[0][j])%M1M2;
            __int128 x2=((ac[2][j]-r1%mod[2])%mod[2]+mod[2])%mod[2];
            x2=x2*invM1M2_modM3%mod[2];
            __int128 r2=(x2*M1M2%M1M2M3+r1)%M1M2M3;
            __int128 x3=((ac[3][j]-r2%mod[3])%mod[3]+mod[3])%mod[3];
            x3=x3*invM1M2M3_modM4%mod[3];
            __int128 temp=x3%p_;
            temp=temp*(mod[0]%p_)%p_;
            temp=temp*(mod[1]%p_)%p_;
            temp=temp*(mod[2]%p_)%p_;
            temp=(temp+r2%p_)%p_;
            ab[j]=(long long)temp;
        }
        //        auto End=std::chrono::high_resolution_clock::now();
        auto End=MPI_Wtime();
//        std::chrono::duration<double,std::ratio<1,1000>>elapsed=End-Start;
        double elapsed=End-Start;
//        ans += elapsed.count();
        ans+=elapsed;
        fCheck(ab,n_,i);
        std::cout<<"average latency for n="<<n_<<" p="<<p_<<" : "<<ans*1e3<<" (us) "<<std::endl;
        // 可以使用 fWrite 函数将 ab 的输出结果打印到 files 文件夹下
        // 禁止使用 cout 一次性输出大量文件内容
        fWrite(ab,n_,i);
    }
```

我们的基本逻辑是：首先将 4 模数的子 NTT 拆分到每个进程中，做正常计

算；而后我们做一个阻塞，来确保广播传输信息时是安全的。

在后面，我们判断是否为主进程，并且通过 recv 和 send 函数做数据的汇集。而最后的 CRT 部分，我们仅在主进程中执行。

可以发现这样做相当于用多进程简单代替了多线程，实现了类似的效果。我们在脚本中申请了 4 个结点，每个结点 1 个线程。以下是我们在服务器上的初步测试结果：

```
ntt > test.o
  多项式乘法结果正确
  average latency for n=4 p=7340033 : 1.36805 (us)
  多项式乘法结果正确
  average latency for n=131072 p=7340033 : 162.981 (us)
  多项式乘法结果正确
  average latency for n=131072 p=104857601 : 161.714 (us)
  多项式乘法结果正确
  average latency for n=131072 p=469762049 : 163.475 (us)
  多项式乘法结果正确
  average latency for n=131072 p=1337006139375617 : 165.652 (us)

  Authorized users only. All activities may be monitored and reported.

```

可以发现顺利执行，并且耗时与多线程大致相当。

于是接下来我们考虑进一步的优化。

首先进行分析。我们目前的策略是：多进程并行地做 1 次 NTT，而后主进程接收广播信息并做 CRT 合并。对于 NTT 部分，由于蝴蝶变换的三层循环不太好分析，再加上内部变量交错可能形成冲突，我们先留待之后优化；而对于 CRT，我们发现这层循环非常简单，相当于一次 n 量级的遍历，并且由于有 int128 的存在效率非常低，所以我们应当着眼于这块去尝试采用 OpenMP 优化。

可以发现四模数合并 CRT 的循环外部已经有：

```
    __int128 M1M2=mod[0]*mod[1];
    __int128 M1M2M3=M1M2*mod[2];


    __int128 invM1_modM2=ksm(mod[0], mod[1]-2, mod[1]);
    __int128 invM1M2_modM3=ksm(M1M2%mod[2], mod[2]-2, mod[2]);
    __int128 invM1M2M3_modM4=ksm(M1M2M3%mod[3], mod[3]-2, mod[3]);
```

并且合并过程中的临时变量都在循环内部定义，于是尽管其非常复杂，我们仍然可以采用默认的并行策略来加速。代码如下：

```
#pragma omp parallel for schedule(static)
        for (int j=0; j <= lim; ++j) {
```

```
            __int128 x1=((ac[1][j]-ac[0][j])%mod[1]+mod[1])%mod[1];
            x1=x1*invM1_modM2%mod[1];
            __int128 r1=(x1*mod[0]%M1M2+ac[0][j])%M1M2;

            __int128 x2=((ac[2][j]-r1%mod[2])%mod[2]+mod[2])%mod[2];
            x2=x2*invM1M2_modM3%mod[2];
            __int128 r2=(x2*M1M2%M1M2M3+r1)%M1M2M3;
            __int128 x3=((ac[3][j]-r2%mod[3])%mod[3]+mod[3])%mod[3];
            x3=x3*invM1M2M3_modM4%mod[3];
```

```
            __int128 temp=x3%p_;
            temp=temp*(mod[0]%p_)%p_;
            temp=temp*(mod[1]%p_)%p_;
            temp=temp*(mod[2]%p_)%p_;
            temp=(temp+r2%p_)%p_;
            ab[j]=(long long)temp;
        }
```

在这里我们初步完成了对其大致的测试。脚本中我们申请了 4 结点 8 线程 4 核心。可以发现和所想的效果类似，这块的加速确实比较显著：

```
ntt > test.o
1   多项式乘法结果正确
2   average latency for n=4 p=7340033 : 5.10788 (us)
3   多项式乘法结果正确
4   average latency for n=131072 p=7340033 : 118.253 (us)
5   多项式乘法结果正确
6   average latency for n=131072 p=104857601 : 123.115 (us)
7   多项式乘法结果正确
8   average latency for n=131072 p=469762049 : 114.468 (us)
9   多项式乘法结果正确
10  average latency for n=131072 p=1337006139375617 : 113.83 (us)
11
12  Authorized users only. All activities may be monitored and reported.
13
```

而后我们着眼于蝴蝶变换部分。

难以加速的一个原因是：我们以前采用多线程方法的时候，关注的重点总是最内层循环。这很好理解，因为最内层循环同步遍历了 mid 规模以及单位根 w 的角度，非常方便我们拆分。但在 OpenMP 中，我们如果不采用更细粒度的指令是没有办法避免 w 在旋转时的冲突的；而如果换用 Pthread 等方法，我们可以知道其效率是不如 OpenMP 的（见上次报告分析部分）。

但是这其实只是一个误区，或者说是盲点。对于 OpenMP 加速而言，我们往往只考虑嵌套的外层循环，而内部的循环就可以当作展开来理解。亦即在保证没有数据竞争的情况下，只需要选择一重循环进行 OpenMP 的指导优化即可。于

是我们可以发现第二重循环满足这个条件，于是有：

```
#pragma omp parallel for schedule(static)
        for(long long j=0;j<lim;j+=(mid<<1)){//主体
            long long w=1;
//            #pragma omp parallel for schedule(static)
            for(long long k=0;k<mid;++k,w=1ll*w*wn%p){
                long long x=a[j+k];
                long long y=a[j+k+mid]*w%p;
                a[j+k]=(x+y)%p;
                a[j+k+mid]=(x-y+p)%p;
            }
        }
```

并且，我们的所有一重简单循环都可以并行化，如下是一个例子，用于初始化每个进程自身的子 NTT 单元：

```
    #pragma omp parallel for schedule(static)
   for(int J=0;J<=lim;++J){
       ac[rank][J]=a[J]%mod[rank];
       bc[rank][J]=b[J]%mod[rank];
   }
```

在做完了所有的 OpenMP 优化后，我们得到了以下初步测试结果：

```
多项式乘法结果正确
average latency for n=4 p=7340033 : 18.5769 (us)
多项式乘法结果正确
average latency for n=131072 p=7340033 : 163.908 (us)
多项式乘法结果正确
average latency for n=131072 p=104857601 : 117.063 (us)
多项式乘法结果正确
average latency for n=131072 p=469762049 : 114.683 (us)
多项式乘法结果正确
average latency for n=131072 p=1337006139375617 : 111.487 (us)

Authorized users only. All activities may be monitored and reported.
```

可以发现尽管相比没有 NTT 内 OpenMP 优化的版本来说，提速并不显著，但在数据变大的时候也略有成效。

至此，我们的全部优化已经完成了。关于进一步的分析，见下面的部分。

四．实验分析

和上次实验类同，我们的分析主要分为两个部分，分别是控制变量 n 和 p，

用于探讨不同维度的数据规模变化下算法的性能。**对比的算法一共有四种，分别是朴素串行 NTT、串行 CRT 大模数四模 NTT、CRT+MPI 实现的多进程 NTT，以及 CRT+MPI+OpenMP 实现的多进程+多线程 NTT。**

由于 MPI 的计时相比之前有了改动，为了控制变量使结果置信度更高，所以我们对于之前测试过的串行 NTT 和串行 4 模 CRT 实现的 NTT，也改用了 MPI 的计时方式来进行重新测试。当然，朴素 NTT 无法实现大模数，所以我们的模数应当控制在一个合理的范围。

接下来，我们将会分别控制变量 n 和 p 来进行分析。关于核心数量等参数所带来的优化，这里不作详细分析，因为它和我们编写的算法没有任何关系。

与之前相似地，以下是基于 perf 工具在本地获得的串行版本 NTT 的不同数据规模性能测试数据，p 取 104857601：

n	时间 1（us）	时间 2（us）	时间 3（us）	时间 4（us）	时间 5（us）	平均（us）
1000	0.563	0.589	0.572	0.596	0.574	0.5788
10000	6.102	5.944	5.871	6.031	5.982	5.986
100000	66.911	65.973	66.218	66.559	65.997	66.3316

表 1 朴素串行耗时数据

以下是 4 模数 CRT 串行 NTT 的性能测试数据：

n	时间 1（us）	时间 2（us）	时间 3（us）	时间 4（us）	时间 5（us）	平均（us）
1000	3.871	3.882	3.871	3.903	3.889	3.8832
10000	40.623	39.972	39.991	40.189	39.877	40.1304
100000	447.338	446.224	447.853	440.932	446.847	445.8338

表 2 CRT 串行耗时数据

下面是 4 进程 MPI 下的 4 模数 CRT 并行 NTT 数据：

n	时间 1（us）	时间 2（us）	时间 3（us）	时间 4（us）	时间 5（us）	平均（us）
1000	1.633	1.634	1.633	1.621	1.623	1.6288
10000	16.308	16.996	16.998	16.421	16.403	16.6252
100000	160.338	163.427	161.486	161.557	167.551	162.8718

表 3 CRT_MPI 耗时数据

下面是 OpenMP 优化下的 MPI 4 模数 CRT 并行 NTT 数据：

n	时间 1（us）	时间 2（us）	时间 3（us）	时间 4（us）	时间 5（us）	平均（us）
1000	20.288	18.569	29.307	19.063	19.308	21.307
10000	54.627	63.118	40.629	39.184	45.422	48.596
100000	159.187	113.628	167.199	116.985	164.091	144.218

表 4 CRT_OpenMP&MPI 耗时数据

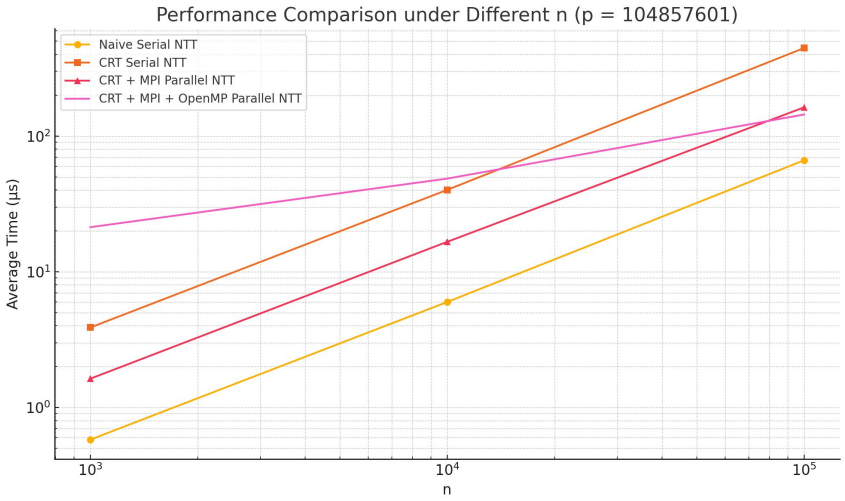

图 1 耗时对比

上面是控制模数不变、耗时随 n 规模而变化的曲线，横纵轴均采用对数刻度。可以看到和上次相似，我们的结果呈现为：未优化的 CRT 4 模 NTT 明显最慢，MPI 的并行 CRT 算法稍快，再慢于普通 NTT，但是三者的常数大约在同一量级；同时 MPI 与 OpenMP 共同优化的 NTT 呈现出先慢后快的趋势。

朴素 NTT 算法的速度快是正常的，这是因为它天然地不支持大模数，从而常数小。这一点在上次的实验报告中已经分析过了，在这里不再赘述。在这里列出它只是为了做出对比，但更多地我们会分析 CRT 下不同方法的优化。

至于 CRT 在使用 MPI 多进程优化前后出现的差异，是很符合直觉的。以朴素 CRT 为基准，**MPI 加速的算法加速比在实验的规模下分别为 2.384、2.413、2.737，稳定在 2.5 倍左右的量级**。不过鉴于我们的进程数为 4，这说明了申请进程的开销并不是可以忽略的。

关于反常的 MPI+OpenMP 优化，经过观察我们可以作如下分析：我们注意到在未进行 NTT 内部 OpenMP 优化时，n=4 的样例中耗时虽然已经膨胀，但还是远远快过实现之后。于是很自然地，我们能够得到 OpenMP 开销过大的结论。而当数据规模逐渐扩大，OpenMP 的开销逐渐可以被忽略，在 10^5 的规模下终于反超了仅进行 MPI 优化的版本。可以相信，当数据规模继续扩大，其有望做到朴素串行 NTT 的速度。这显然是可以接受的：因为我们的单模 NTT 仅仅支持 1e9 内的极少模数，灵活性非常差。这点我们在上次也讨论过了，不再展开。

关于两种优化的具体区别，我们先做完对 p 变量的控制实验再一并讨论。

与上次同样地，以下是取不同规模模数，并控制 n=131072 进行实验的耗时数据。**和上次一样，由于前面的单模 NTT 不支持大模数，笔者在本地加入了 int128 暴力扩充了其支持范围，并且也同样使用 MPI 计时方式和编译方式，**来方便对比分析。并且由于小模数区别太过微小，故从 1e6 规模开始测试。以下是具体测试数据：

p	时间 1（us）	时间 2（us）	时间 3（us）	平均（us）

7340033	555.923	556.305	556.197	556.142
469762049	557.832	557.392	557.705	557.643
77309411329	560.341	560.768	560.352	560.487
6597069766657	564.072	564.419	564.149	564.213
263882790666241	570.142	570.891	568.942	569.992
7881299347898369	573.487	573.207	573.223	573.306
180143985094819841	583.047	585.340	584.217	584.201

表 5 暴力串行 NTT（int128）耗时数据

p	时间 1（us）	时间 2（us）	时间 3（us）	平均（us）
7340033	450.013	450.572	449.906	450.164
469762049	450.018	450.760	450.235	450.338
77309411329	452.389	451.732	451.897	452.006
6597069766657	435.444	452.710	453.198	453.117
263882790666241	456.178	455.392	455.868	455.813
7881299347898369	457.466	456.912	456.931	457.103
180143985094819841	460.592	461.378	460.019	460.996

表 6 CRT 串行 NTT 耗时数据

p	时间 1（us）	时间 2（us）	时间 3（us）	平均（us）
7340033	159.822	160.502	160.087	160.137
469762049	162.617	161.738	162.339	162.231
77309411329	163.892	163.217	163.818	163.642
6597069766657	165.938	165.211	165.483	165.544
263882790666241	165.903	166.530	165.879	166.104
7881299347898369	167.641	168.324	167.732	167.899
180143985094819841	169.281	168.437	169.001	168.903

表 7　CRT_MPI 耗时数据

p	时间 1（us）	时间 2（us）	时间 3（us）	平均（us）
7340033	109.827	124.522	117.690	117.346
469762049	117.135	134.389	122.901	124.808
77309411329	110.784	115.128	117.500	114.471
6597069766657	106.391	118.654	112.884	112.643
263882790666241	110.173	122.206	116.577	116.319
7881299347898369	121.830	110.255	113.612	115.232
180143985094819841	112.491	128.183	119.818	120.164

表 8 CRT_MPI&OpenMP 耗时数据

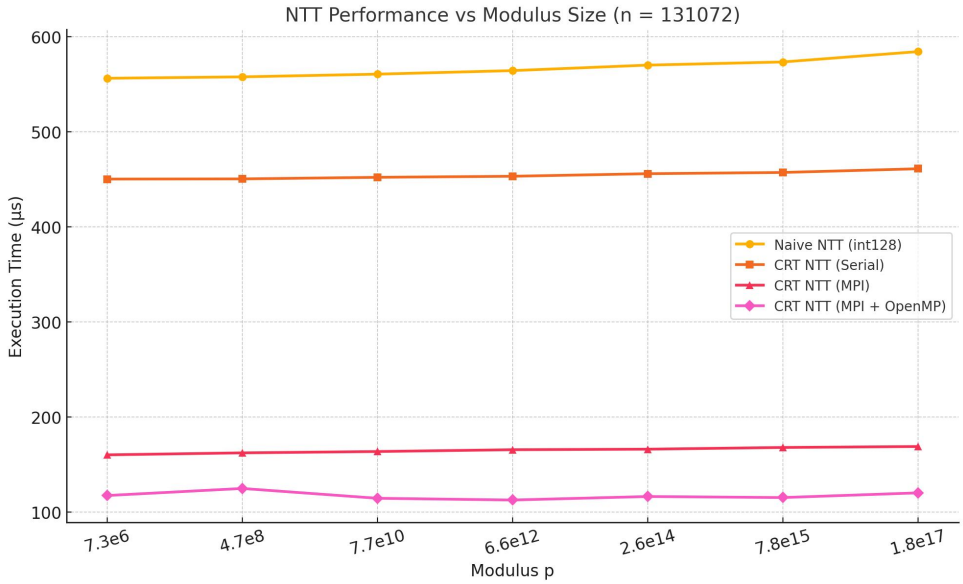

图 2 耗时对比

和上次的结果相似，采取暴力 int128 的朴素 NTT 来到了最慢的位置，而采用 CRT 合并的 NTT 相比它而言加速比与上次差不多一致，这里不再赘述。

让我们继续分析 MPI 优化以及 MPI+OpenMP 优化。**前者相比朴素 NTT 加速比为 3.4 左右，相比串行 CRT 加速比也达到了 2.74 左右；而后者相比朴素 NTT 达到了 4.8 以上的加速比，相比于串行 CRT 也达到了 3.8 左右的加速比。**这个数据表明我们的优化总体上是非常成功的。

我们在上次的报告中已经分析过了，int128 的常数十分吓人，在模乘下的 int128 会比正常慢 6 到 10 倍，而这就是暴力 NTT 的常数所在。而相比前者，CRT 仅仅在合并时采用了 int128，是一个串行的上位替代。这些和上次的分析相似，这里就不再多说了。

接下来，我们进行理论分析的对照。

仅使用 MPI 的 4 模 NTT 中，我们申请了 4 个进程，和多线程类似，所以理论的优化比是 4.0。而实际实验中，相比串行 CRT 的加速比仅有 2.74，可以被解释为多进程间通信以及整体的初始化开销过大。

而在 MPI 基础上优化的 OpenMP 版本中，理论上 4 个进程有 8 个线程，加

速比远远大于实际。可以推测出 OpenMP 优化的部分更近似于“蚊子腿”的部分（主要并行部分已经由 MPI 来完成了），所以开销的占比很大，导致优化并不显著。不过总体还是有比较大优化的，特别是模数较大的时候。

一个反常的结果是：在最后一种版本中，我们的数据波动较大，并且其波动甚至有时盖过了 p 规模带来的影响。这可以从两方面来分析：首先以 CRT 为基础的方法在模数拆分的规模上都相差不大，本身受 p 的影响就并不大；其次，MPI 和 OpenMP 毕竟是两个层面的并行方式，可能在兼容中并不能完全配合。MPI 通信的干扰可能 OpenMP 并行：在某些阶段，线程之间共享内存时，MPI 进程之间仍需通过网络通信；若 MPI 通信阻塞，会导致 OpenMP 内部线程无法顺利执行，进而引起性能抖动。

关于与 SIMD 结合的可能，在这里也进行探讨。**笔者先给出结论：至少和目前的 CRT NTT，它是没有办法做到高效结合的。**

首先 SIMD 的向量化操作看起来与 OpenMP 的多线程策略互斥性是比较强的，究其原因是 OpenMP 本身就是对于循环所进行拆分的并行，而 SIMD 则需要精确地对循环中的元素进行 2 个或 4 个一组的配对。我们诚然可以在配对之后再进行多线程的加速，但是由于我们第一次实验中对 SIMD 的分析里已经发现其优化大约只有 10%以内的量级，如此拆分进行更细粒度并行的开销和优化的幅度相比如何，还未可知也。所以二者是不太好结合的。

但是我们能否不用 OpenMP，而仅换用 SIMD 呢？只用后者会不会得到更优的结果呢？笔者本来是希望进行一些代码实现后的比较的，但稍加分析就会知道：我们的 CRT 中用到了 int128，而这在寄存器中占据的长度过大，导致了要么需要进行高低位拆分，要么需要降低并行的向量个数，从而导致这一举措是完全不现实的；而 NTT 蝴蝶变换中如果采用 SIMD，实际上也就像我们前面提到的“蚊子腿”一样，频繁进行向量的迁移所耗费的开销，并不一定比优化带来的效果更大。

而事实上，我们可以根据以前的实验进行更直观的分析。第一次实验中 SIMD 的优化就是基于对蝴蝶操作的向量化——当时在未扩充有效范围的情况下（使用 int），在 n 约等于 10^5 的规模下加速比仅仅为 1.09；当时我们是对 int 的数据分

为 4 个一组，中间步骤短暂拆分为了 2 个一组的形式（出现乘法在 long long 范围内，避免溢出）来进行计算。而当我们使用 long long 作为基数时，**这一步就会直接出现 int128**。我们暂不论它本身带来的性能影响（尽管已经很巨大了），就向量化而言，它的效率就最多只有原来的一半（因为我们的数据字长变为了两倍，向量存储的个数也变为了一半）——这和我们前面分析 CRT 部分是否可以 SIMD 时的结论是完全一致的。**鉴于原先的加速比仅为 1.09，在减半后极大概率是会产生负优化的。所以我们能下结论：至少笔者所实现的 CRT NTT 并行优化中，是不适合用 SIMD 的。**

进一步地，我们来进行 MPI 不同划分策略的探讨。

在我们的实现中，做到的是基于数据划分的 MPI 并行。它的结构十分清晰，所以在一开始就很容易让人想到。不过它的劣势在于通信开销随进程数上升，以及主进程 CRT 的合并在串行层面上耗费时间过多，这是可优化的部分，所以我们才有后续的 OpenMP 等多线程策略。但我们现在讨论的是 MPI 本身，所以这个瓶颈显然是不可忽略的。

那么我们能否进行任务并行呢？比如将 CRT 部分和 NTT 主体部分拆分，进行并行？这是不太现实的。因为 CRT 计算所需的是 NTT 的结果，本身就是有时效性的；而如果我们强行拆开并且阻塞等待，很显然和串行是没有太大区别的。

或许有人会认为，基于任务并行思想可以进行更细粒度的拆分，从而将能够并行的任务同时进行计算——这实际上就是流水线思想。但它的可行性可以从以下两个角度反驳：首先是这个粒度的问题。**因为 NTT 是一个递归的过程，我们是很难找到其子任务，使得它和外界可以被拆分开的。**其次就是通信开销的问题：更细粒度的并行意味着更多次数、更多不同位置的通信，**而通信开销是较为巨大的**——哪怕我只进行 O(n)量级的通信，根据 $\mathcal{O}(\tau+\sigma m)$ 的公式[1]，也会有 O(σn) 的开销——这在总体为 nlogn 量级的算法中已经是难以忽略的了。

所以我们的结论是：最简单的数据划分 MPI，同时也是最有效的。而其他策略和 SIMD 一样，在我们的算法中是没有正向贡献的。

以上就是我们的性能分析和理论辨析部分。综上我们的任务已经完成。

五．总结

本文基于 MPI 编程技术，针对 4 模数 CRT 下的 NTT（MTT）算法进行了系统的并行优化设计。通过结合 MPI 和 OpenMP 多进程+多线程策略，实现了极小常数的任意模数 NTT。

实验中，笔者详细分析了实验目标，并且系统探讨了为何采用数据并行而非其他 MPI 策略；同时也排除了 SIMD 这一不优的工具。在实现的不同阶段，笔者也保留了不同版本的代码进行性能测试，得出了比较可信而直观的加速比实验结果。

在实验过程中，笔者在获得了 MPI 编程宝贵经验的同时，也深入体会到了多种并行策略结合的思想，对并行程序设计有了较为深入的理解。

六．项目链接

https://github.com/Hsy23333/NTT_parallel

七．参考文献

[1] “HBJ model,” *Wikipedia*, The Free Encyclopedia, 15 Sep. 2024. [Online]. Available: https://en.wikipedia.org/wiki/HBJ_model?utm_source=chatgpt.com. [Accessed: 14 Jun. 2025].