Pthread&OpenMP 编程：基于 CRT 的 NTT 优化

2313911 黄尚扬

2025.5.20

目录

一. 引言 3
二. 问题描述 3
三. 算法设计 4
四. 实验分析 12
五. 总结 18
六. 项目链接 18
七. 参考文献 18

一．引言

pthread（POSIX threads）和 OpenMP（Open Multi-Processing） 是两种主流的并行编程技术，用于在多核处理器上实现任务并行与数据并行，从而提升计算性能。pthread 提供了底层线程控制机制，允许程序员对线程的创建、同步和调度进行精细化管理；而 OpenMP 以其基于编译指令的高层抽象，便于开发者快速实现并行化，特别适用于循环级别的任务并行。它们被广泛应用于科学计算、工程模拟、图像处理、大规模数据分析等需要高性能计算的领域。

多项式乘法 作为一种基础数学运算形式，在信号处理、计算机图形学和密码学中占据重要地位。在本并行课题中，我们聚焦于 同态加密中的多项式乘法。同态加密允许在密文上直接进行数学运算，解密后可得到与明文运算一致的结果，这在保障数据隐私和安全计算中具有重要意义。而高效的多项式乘法是实现同态加密性能提升的关键路径之一。

本研究基于 pthread 和 OpenMP 框架，针对现有的 NTT 算法实现并行优化，旨在充分利用多核处理器资源，加速多项式乘法的计算过程。

二．问题描述

本次多线程实验分为三个方案,朴素算法,四分算法和 CRT 算法。朴素算法为基础要求,后两个为进阶要求。笔者将会着手进行**朴素算法和 CRT 算法**的实现。

关于简单的 NTT 本身，笔者在 SIMD 编程的部分已经较为详细地阐述了。这里先主要谈谈笔者任意模数 NTT（MTT）的理解，方便进行下面实验。

首先我们有 CRT 算法，亦即“中国剩余定理”。[1]中国剩余定理 (Chinese Remainder Theorem, CRT) 可求解如下形式的一元线性同余方程组（其中 $n_1, n_2, \cdots, n_k$ 两两互质）：

$$\begin{cases} x & \equiv a_1 \pmod{n_1} \\ x & \equiv a_2 \pmod{n_2} \\ & \vdots \\ x & \equiv a_k \pmod{n_k} \end{cases}$$

关于它的具体实现，相信打过 OI 的人都十分清楚。由于它不是实验重点，我们着重强调它在 MTT 中的意义而不是做法。

考虑到平凡 NTT 对模数的要求十分苛刻，对于任意的数我们常常无法直接进行快速卷积。此时我们需要选取已知的多个合法模数（通常为 3 个）分别进行 NTT，得到三个结果序列；最后利用上述同余方程对结果的每一项进行合并，得到方程通解。这一过程我们采用 excrt。

由于最后的序列是已经合并过的结果，我们对其直接暴力模 p（也就是要求的模数）即可。这里对 p 没有要求。

为什么这样做是正确的？笔者做如下理解：模运算会构建出一个循环群，而多个互质的数循环节会出现在其乘积之外。**所以我们在使用 excrt 求解方程的时候，可以保证在所有小模数乘积之内有且仅有一个解；而很显然当我们的计算本身就在小模数乘积范围内时，求出来的这个解就是子 NTT 意义下的解。**只要 p 也落在这个范围内，我们就可以通过模 p 等价地做对 p 的 NTT。

于是我们就可以用常数次 NTT 操作对任意模数进行兼容。这就是 MTT 的原理。

我们注意到这里实际上执行了完全相同的多次操作，于是对于每个子 NTT，我们可以分配一个线程，实现并行的优化。下面我们准备进行实际的算法编写。

三．算法设计

我们首先实现朴素的优化。**由于笔者的重点在 CRT，所以朴素的地方并没有多想，可能写得较为暴力**，最后导致了负优化。但优化的思路应当是没有太大问题的。

对于蝴蝶操作的最内层部分，也就是下面这个地方，我们很显然可以进行拆分。

```
long long w=1;
  for(long long k=0;k<mid;++k,w=1ll*w*wn%p){
      long long x=a[j+k];
      long long y=1ll*a[j+k+mid]*w%p;
      a[j+k]=(x+y)%p;
      a[j+k+mid]=(x-y+p)%p;
  }
```

一个很显然的思路是，遍历 a 数组的时候是线程安全的，所以我只要拆成几

个部分分别进行计算即可。

为此我们有：

```
pthread_t ths[4];//朴素优化，使用 4 个线程
struct qwqqq{//封装参数，传给线程
    long long mid,wn,j,p;
    int KK;//KK=0,1,2,3
    long long *a;
}tmp4[4];//创建 4 个暂时参数结构体
void* pth(void* tmpw){
    qwqqq* tmpq=(qwqqq*) tmpw;
    long long mid=tmpq->mid;
    long long wn=tmpq->wn;
    long long j=tmpq->j;
    long long p=tmpq->p;
    long long w=ksm(wn,tmpq->KK,p);//不同的 KK 值对应起点不同
    long long wn4=ksm(wn,4,p);//改为转 4 下
    long long *a=tmpq->a;
    for(long long k=tmpq->KK;k<mid;k+=4,w=1ll*w*wn4%p){
        long long x=a[j+k];
        long long y=1ll*a[j+k+mid]*w%p;
        a[j+k]=(x+y)%p;
        a[j+k+mid]=(x-y+p)%p;
    }
    return NULL;
}
```

上面的代码实现了参数的封装和线程、函数的定义。内部的循环根据起点不同来进行区分，每次旋转 4 倍单位根。关于线程的创建和初始化，我们有：

```
            //下面是朴素优化
            for(int v=0;v<4;++v){
                tmp4[v].j=j;
                tmp4[v].KK=v;
//                tmp4[v].mid=mid;
//                tmp4[v].p=p;
//                tmp4[v].wn=wn;
                int errcode=pthread_create(&ths[v],NULL,pth,&tmp4[v]);
            }
            for(int v=0;v<4;++v)    pthread_join(ths[v],NULL);//等待这层所有线程结束
```

这个部分是放在第二层循环内部的，简单替代了原先的平凡操作。上面的参数初始化中，我们注释掉了三行；但实际上是放在了外部的相应循环中，来小小地进行了优化，防止反复冗余赋值。

这份代码是反向优化的，在服务器上测试，发现其速度并不及朴素 NTT。~~（但毕竟它能跑，能跑起来就是好代码）~~**不过在结果上，笔者已经实现了朴素优化部分**。关于负优化的原因，具体分析见下一个模块。**这里笔者先基于大模数 NTT 着手实现 CRT 优化。**

我们先来试图实现一下基本的三模数合并 NTT，用以通过大模数样例。一上来写出的版本重点部分如下：

```
//下面是三模数 ntt
 /**/  for(int i=0;i<3;++i){
       memcpy(ac[i],a,sizeof(a));
       memcpy(bc[i],b,sizeof(b));
       ntt(ac[i],bc[i],ac[i],n_,mod[i]);
   }//分别执行子任务
   long long M1M2=mod[0]*mod[1];
   long long invM1_modM2=ksm(mod[0],mod[1]-2,mod[1]);            //inv(mod[0]) mod mod[1]
   long long invM1M2_modM3=ksm(M1M2%mod[2],mod[2]-2,mod[2]);     // inv(mod[0]*mod[1]) mod
mod[2]

   for (int i=0;i<=lim;++i) {
       long long x1=((ac[1][i]-ac[0][i])%mod[1]+mod[1])%mod[1];
       x1=x1*invM1_modM2%mod[1];
       long long r1=(x1*mod[0]%M1M2+ac[0][i])%M1M2;

       long long x2=((ac[2][i]-r1%mod[2])%mod[2]+mod[2])%mod[2];
       x2=x2*invM1M2_modM3%mod[2];
       ab[i]=(x2*(mod[0]%p_)%p_*(mod[1]%p_)%p_+r1%p_)%p_;
   }//crt 合并
```

我们新建了 6 个临时数组来存储 3 模数分别的结果，最后使用 CRT 将其合并。

笔者原本认为这份代码已经没有什么问题，<u>毕竟是从我自己在洛谷模板题的 AC 提交稍加更改而来的</u>。但是申必的事情发生了：大模数的样例在很长一段时间内都无法通过。**然而前几个样例都能通过，说明合并的逻辑是没有问题的，只可能是操作的过程中出现了申必的溢出。**在经过两个心态爆炸的晚上后，笔者甚至在最后的合并中使用了 int128，然而没有用处。

那么，我们先来思考一下问题所在。

首先，我们的 ntt 部分应当是没有问题的。因为我们已经将大模数的问题变为了 1e9 以内小模数的问题，<u>并且我们在 debug 中已经将上面的 memset 改为循</u>

环赋值取模避免初始值溢出，所以这和普通的 ntt 并没有逻辑上的区别，溢出并不会在 ntt 中出现。

那么，是模数的问题吗？我们使用的三个模数都是原根为 3 并且数量级约等于 1e9 的模数，根据笔者一开始的理解，其乘积大于模数的 1e15，似乎是可行的。但果真如此吗？

笔者通过查阅资料（由于调试得心态比较爆炸再加上看的网页文献有点多所以没有记下来具体的引用地址，此处留待补充）发现，先前的分析漏掉了输入数据中间操作的溢出问题，亦即 $AB > p_1p_2p_3$ 的问题。我们代入现有的模数，可以发现{1004535809,1998585857,998244353}的乘积在 1e27 级别，而输入数据的平方在 1e30 级别，很显然在这里爆炸了。

那么我们应该如何解决？笔者作如下思考：

我们首先肯定不希望改变太多框架内容，所以优先保持三模数现状。那么，我们就得将所取模数扩大。很显然，我们一定会用到 1e10 以上规模的模数，而这会带来两个问题：

其一就是 ntt 内部的溢出问题。我们的朴素 ntt 本身是为了 int 而设计的，其乘积最多也不超过 longlong，对其作强制转换再取模可以很轻松地变回 int；但是如果 ntt 操作数据本身就大于 int 范围，我们就需要考虑频繁使用 int128，而这带来的时间代价非常大。

其次，我们不妨思考一下为何要使用多模数+CRT 的方式实现 NTT。我们不正是难以对大于 int 范围的模数操作，所以才作拆分吗！在这种情况下，我们如果滥用 int128 进行暴力操作，这似乎还不如直接采用单模数。

所以综上所述，三模数的 CRT 在这里是不合理的。我们考虑变为四模数。

以下是我们的四模数 NTT 核心代码，模数采用{1004535809,1998585857,998244353,469762049}：

```
//下面是多模数 ntt
  /**/  for(int j=0;j<4;++j){
        //memcpy(ac[j],a,sizeof(a));
        //memcpy(bc[j],b,sizeof(b));
        for(int J=0;J<=lim;++J){
            ac[j][J]=a[J]%mod[j];
            bc[j][J]=b[J]%mod[j];
        }
```

```
        ntt(ac[j],bc[j],ac[j],n_,mod[j]);
    }//分别执行子任务
    __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;
    }
```

下面是我们运行的结果，可以看到大模数的样例顺利通过：

```
多项式乘法结果正确
average latency for n=4 p=7340033 : 0.03693 (us)
多项式乘法结果正确
average latency for n=131072 p=7340033 : 447.338 (us)
多项式乘法结果正确
average latency for n=131072 p=104857601 : 448.282 (us)
多项式乘法结果正确
average latency for n=131072 p=469762049 : 449.923 (us)
多项式乘法结果正确
average latency for n=131072 p=1337006139375617 : 454.419 (us)
```

实际上，我们一共执行了 12 次 NTT 操作（3 次乘四组），所以实际效率肯定是要慢于普通 NTT 的。**但就目前而言，这个拆数的作用并不在优化，而在于实现大模数或者任意模数的卷积处理。接下来的优化才是我们的重点。**

在前面实现的大模数 NTT 当中，很显然我们有两个很明显的循环。在上面

的循环里，我们可以简单地创建 4 线程来实现并行加速；而后面的 CRT 合并部分也是线程安全的，所以我们一样可以分多线程进行操作。我们可以预估，在多线程的优化下，我们所耗费的时间将会是原先朴素四模数 NTT 的 1/4，从而在扩充模数范围的情况下实现与一般 NTT 相似规模的时间复杂度。

具体的线程执行操作和朴素多线程写法几乎一致，这里就不展开了；而不同的细节操作部分如下：

```
pthread_t ths[4];//考虑对 4 模数 NTT 并行优化
struct qwqqq{//封装参数，传给线程
    long long n_;
    int j;//0,1,2,3
}tmp4[4];//创建 4 个暂时参数结构体
struct awaaa{
    __int128 M1M2,M1M2M3,invM1_modM2,invM1M2_modM3,invM1M2M3_modM4;
    int K;
    long long p_;
}tmpp4[4];
void* NTT_parallel(void* tmpw){
    qwqqq* tmpq=(qwqqq*) tmpw;
    int j=tmpq->j;
    long long n_=tmpq->n_;
    for(int J=0;J<=lim;++J){
        ac[j][J]=a[J]%mod[j];
        bc[j][J]=b[J]%mod[j];
    }
    ntt(ac[j],bc[j],ac[j],n_,mod[j]);
    return NULL;
}
void* CRT_parallel(void* tmpw){
    awaaa* tmpq=(awaaa*) tmpw;
    __int128 M1M2=tmpq->M1M2;
    __int128 M1M2M3=tmpq->M1M2M3;
    __int128 invM1_modM2=tmpq->invM1_modM2;
    __int128 invM1M2_modM3=tmpq->invM1M2_modM3;
    __int128 invM1M2M3_modM4=tmpq->invM1M2M3_modM4;
    long long p_=tmpq->p_;
    int K=tmpq->K;

    for (int j=K; j <= lim; j+=4) {
        __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;
    }
    return NULL;
}
```

下面是我们得到的结果，和我们的预估相似，**到这里为止我们已经完成了 pthread 下的大模数 NTT 并行优化，也就是我们的进阶要求：**

```
多项式乘法结果正确
average latency for n=4 p=7340033 : 0.338971 (us)
多项式乘法结果正确
average latency for n=131072 p=7340033 : 159.13 (us)
多项式乘法结果正确
average latency for n=131072 p=104857601 : 160.484 (us)
多项式乘法结果正确
average latency for n=131072 p=469762049 : 160.936 (us)
多项式乘法结果正确
average latency for n=131072 p=1337006139375617 : 164.465 (us)
```

接下来我们简要进行 openmp 的实验，同样是基于 4 模数拆分。代码非常简单：

```
//下面是多模数 ntt
 #pragma omp parallel
 {
    #pragma omp for
    for(int j=0;j<4;++j){
        for(int J=0;J<=lim;++J){
            ac[j][J]=a[J]%mod[j];
            bc[j][J]=b[J]%mod[j];
        }
        ntt(ac[j],bc[j],ac[j],n_,mod[j]);
```

```
    }//分别执行子任务
 }


    __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]);
#pragma omp parallel shared(p_)
{
    #pragma omp for
    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;
    }
}
```

初步结果如下：

```
AttributeError: module 'version' has no attribute 'VERSION'
多项式乘法结果正确
average latency for n=4 p=7340033 : 0.621521 (us)
多项式乘法结果正确
average latency for n=131072 p=7340033 : 105.142 (us)
多项式乘法结果正确
average latency for n=131072 p=104857601 : 105.379 (us)
多项式乘法结果正确
average latency for n=131072 p=469762049 : 105.289 (us)
多项式乘法结果正确
average latency for n=131072 p=1337006139375617 : 105.952 (us)
```

至此，我们的全部进阶要求已经完成了。关于进一步的分析，见下面的部分。

四．实验分析

我们的分析主要分为两个部分，分别是朴素优化多线程 NTT 和原先的对比分析，以及任意模数下的 NTT 优化前后对比分析。**第二部分是我们的重点。**

我们先进行第一部分的分析。

在前面我们提到，笔者的朴素优化其实是非常暴力的，所以得到了**明显的负优化**。由于负优化是如此显然，我们可以直接跳过重复实验，直接来进行负优化的原因分析。

由于建立线程的成本很高，我们在最内部循环里实际耗费了 O(n)级别的算力来做这个操作，反映到测试用例里就是 1e5 的规模。通过查阅资料，我们可以获知每次线程创建操作大约相当于 1e5 到 1e6 条语句规模。[2]**如此乘起来，它的速度甚至在实际上达到了 $O(n^2)$！**分析的结果很符合我们实际测试时跑起来的速度——约为 1e5us，甚至慢于暴力相乘的 9e4us！

基于本地上的 perf 工具分析，我们也可以发现线程中调用的函数 CPU 占用占比 90%以上，这很符合我们的推断。同时我们也可以进一步推断，当 n 扩大时，优化的作用会逐步显现——但仍然并不快。

接下来我们进入重点的第二部分。

由于这里实现了比较大的模数算法，很多 int 部分被替换为了 long long，笔者对上次的串行 NTT 也重新进行了测试，以确保在算法以外的部分没有差异。

接下来，我们将会分别控制变量 n 和 p 来进行分析。

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	0.576	0.558	0.593	0.575	0.581	0.5766
10000	6.018	5.983	5.990	5.792	5.962	5.949
100000	65.603	64.906	65.990	67.233	66.081	65.9626

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	445.8388

表 2 CRT 串行耗时数据

下面是 pthread 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	152.607	151.032	152.213	152.104	151.931	151.9774

表 3 CRT_pthread 耗时数据

下面是 OpenMP 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	0.937	0.942	0.943	0.937	0.936	0.939
10000	10.903	11.377	10.905	10.914	10.901	11.000
100000	105.371	105.952	105.973	105.896	105.901	105.8186

表 4 CRT_OpenMP 耗时数据

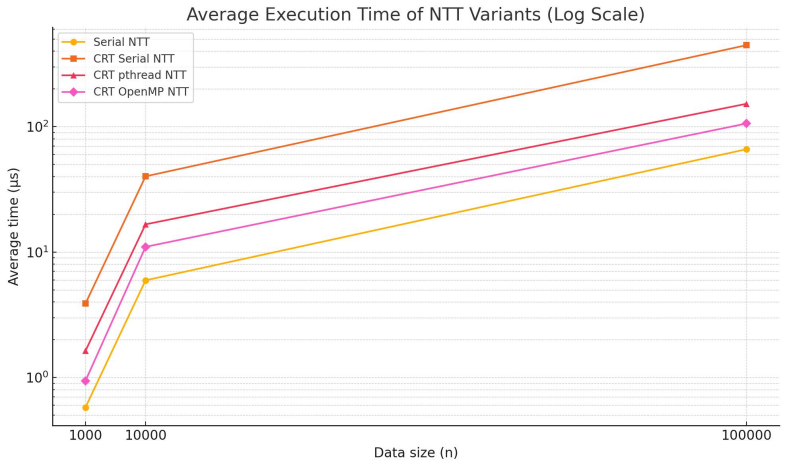

图 1 耗时对比

上面是控制模数不变、耗时随 n 规模而变化的曲线，纵轴采用对数刻度。可以看到，我们的结果呈现为：未优化的 CRT 4 模 NTT 明显最慢，而 pthread 慢于 OpenMP，再慢于普通 NTT，但是三者的常数大约在同一量级。

经过观察，我们作如下分析：由于在实现大数 NTT 的过程中我们使用了 O(n) 级别的 int128 操作来 CRT 合并结果（否则会溢出），我们尽管实现了比较高效率的并行操作，其最终效率和普通 NTT 有小常数级差距是可以理解的。**并且随着 n 的变大，合并带来的开销占比会逐渐缩小，使得 pthread 和 OpenMP 的效率贴近单模 NTT，并且获得接近 4 倍于串行 CRT 的效率。**

不过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。**而在做完多线程优化后，仅仅是多花费了一个 O(n)级别的操作，我们的 NTT 就可以支持 1e18（也就是当四模数都在 1e9 规模时的安全操作范围）以内的所有模数。**当然，如果愿意更改扩充 NTT 函数内部的数据范围，它还能扩充得更大——不过所带来的代价也更大：经过笔者测试，扩容到 1e36 级别将会有爆炸的常数，达到 4 倍多于普通串行的耗时。不过这并不在我们实验的主要考虑范围内，仅仅是一笔带过。

关于 pthread 和 OpenMP 两种优化的区别，我们先做完对 p 变量的控制实验再一并讨论。

以下是取不同规模模数，并控制 n=131072 进行实验的耗时数据。**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，**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	531.207	530.196	530.343	530.582
469762049	533.984	533.990	533.992	533.989
77309411329	536.801	536.794	536.795	536.797
6597069766657	542.329	542.328	542.331	542.329
263882790666241	549.030	549.041	549.032	549.034
7881299347898369	554.747	554.738	554.737	554.741

180143985094819841	563.211	563.243	563.240	563.231

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	447.397	447.338	447.340	447.358
469762049	449.936	449.933	449.911	449.926
77309411329	450.618	450.622	450.623	450.621
6597069766657	452.357	452.333	452.341	452.343
263882790666241	454.699	454.683	454.688	454.690
7881299347898369	456.243	456.278	456.276	456.266
180143985094819841	459.863	459.892	459.881	459.879

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	151.962	151.969	151.960	151.964
469762049	152.338	152.316	152.331	152.328
77309411329	152.551	152.560	152.582	152.564
6597069766657	152.698	152.679	152.711	152.696
263882790666241	152.995	152.994	152.995	152.995
7881299347898369	153.200	153.208	153.199	153.202
180143985094819841	153.476	153.498	153.489	153.488

表 7　CRT_pthread 耗时数据

p	时间 1（us）	时间 2（us）	时间 3（us）	平均（us）
7340033	105.147	105.143	105.138	105.143
469762049	105.288	105.290	105.290	105.289
77309411329	105.316	105.327	105.340	105.328
6597069766657	105.589	105.590	105.571	105.583
263882790666241	105.871	105.921	105.886	105.893
7881299347898369	106.009	106.010	106.005	106.008
180143985094819841	106.231	106.304	106.233	106.256

表 8 CRT_OpenMP 耗时数据

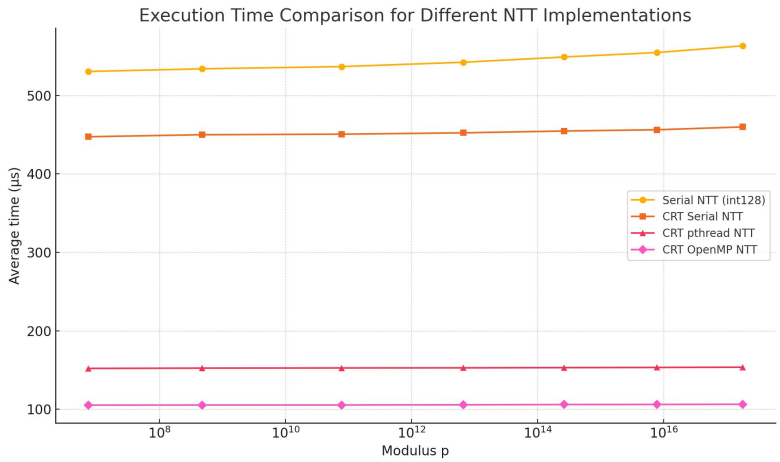

图 2 耗时对比

可以看到，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 相比它有了 15.7%（小模数）到 17.1%（大模数）的效率提升。

让我们继续分析两种多线程优化：**pthread 获得了相比暴力 NTT 71.4%到 72.6%的巨量优化，同时比普通 CRT 快了 66%左右；而 OpenMP 更是夸张，相较于暴力 NTT 快了 80.2%到 81.1%，同时比普通 CRT 快了大约 76.8%！**这和我们先前分析的 4 模数并行结果相似。

让我们进一步分析：由于耗时主要由数据规模而不是模数决定，所以模数仅仅确定了一个常数。很显然 int128 的常数十分吓人，经过笔者对资料的查阅[3]，我们可以知道**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，所以常数较小，并且支持范围较广，是一个完全更优的选择；在 CRT 基础上的两种多线程优化，也沿袭了这一优势。

同时，我们注意到随着 p 的增加，暴力方法耗时有明显的抬头趋势，而 CRT 下的三种方案则变化不大。这和实现原理有关：CRT 仅仅是做出等同规模的拆分，最后再进行合并，只有在合并操作下涉及大数的运算，所以受 p 的影响并不大；而暴力 NTT 直接拿 p 进行操作，自然会深受影响。

关于 OpenMP 和 pthread 的性能差异，笔者进行了一定的资料查阅。具体分析如下：

首先，OpenMP 使用编译器指令（如 #pragma omp parallel）来管理线程，编

译器会在编译时生成线程池并处理线程的创建、销毁和调度，从而减少了运行时的开销。相比之下，pthread 需要手动创建和管理线程，每次调用 pthread_create 都会涉及系统调用，增加了线程管理的开销。[4]而这种开销所带来的时间耗费在 n=1e5 的量级下是不可忽略的，即是造成性能差异的首要原因。

其次，OpenMP 提供了多种任务划分策略，可以根据任务的特性自动将工作负载均匀地分配给各个线程，从而实现更好的负载均衡。pthread 需要程序员手动划分任务，容易导致负载不均衡，影响性能。

在本地 perf 的分析中，pthread 的四线程负载常常出现{21.7%,25.1%,26.0%,27.2%}类似的负载率，这很大程度上是由于小模数规模不同加上手动调用线程并不灵活所导致的；而 OpenMP 就相对均衡。这也印证了上面的推断。

我们具体的实验到这里其实已经完成了，接下来是笔者对采用多线程进行进一步优化的可能思考。

由于 NTT 内部操作的范围限制，CRT 的小模数不超过 1e9 规模是比较好的——**但这并不代表我们一定要尽量取大**。我们在前面所实现的算法中之所以卡着 1e9 左右的规模，是为了避免 CRT 中需要做 5 模数甚至 6 模数操作所带来的代码量；**但实际上为了追求性能，在实现相似规模大数 NTT 目标的前提下，我们完全可以增加小模数的数量，并且压低其大小**。这么做有几个显著的好处：

其一，在小模数足够小的情况下，我们可以保证 CRT 运算不超过 longlong 范围，从而避开 int128 所带来的巨量开销，而多做的几轮 CRT 合并与之相比并不会更慢；

其二，对于每个小模数，我们采取多线程的并行策略，所以总体耗时相当于线程的最长耗时；而显然模数越小，每个线程常数越小，耗时也越小，如此优化可以使得多线程的部分本身更快。

当然，这是基于 n 在 1e5 规模下的思考，所以我们需要保证模数仍然支持 2^{18} 以上的长度，在选取模数的时候需要注意。

但是我们注意到，在这一长度规模下最小的合法模数已经达到了 167772161 的大小，所以我们根本没有选择的空间。那么我们可以考虑基于 2 的幂次进行分块卷积，而小的部分我们可以用短 NTT 来操作。[5]

事实上，这种方法本身就可以拿来做 NTT 的并行优化，但是同样地，我们可以在这基础上继续嵌套 CRT 来沿用我们之前的思路。由于代码过于复杂，在这里笔者仅仅是做一个可行性的思考。

以上就是我对实验的分析以及对进一步优化的思考。

五．总结

本文基于 pthread 和 OpenMP 两种并行编程技术，针对任意模数下的多模数 NTT（MTT）算法进行了系统的并行优化设计。通过结合中国剩余定理（CRT），实现了将大模数问题拆解为多个小模数 NTT 子问题，并行计算后合并结果的高效策略。

实验中，笔者采用 4 线程并行化处理蝴蝶运算，实现了多核环境下的计算加速。针对传统三模数 CRT 合并过程中出现的溢出问题，本文采用了四模数 CRT 方案以扩大数值范围，有效避免中间计算的溢出风险，保证计算的正确性和稳定性。

在实验过程中，笔者获得了 pthread 和 OpenMP 编程的宝贵经验，也对任意模数 NTT 的实现有了进一步的理解。

六．项目链接

https://github.com/Hsy23333/NTT_parallel

七．参考文献

[1] OI Wiki, "中国剩余定理 (CRT)," OI Wiki, https://oi-wiki.org/math/number-theory/crt/ (accessed May 19, 2025).
[2] Ars Technica OpenForum, "Thread creation overhead," Ars Technica, https://arstechnica.com/civis/threads/thread-creation-overhead.174053/ (accessed May 23, 2025).
[3] F. Biscani, "Integer benchmarks," mp++ 2.0.0 Documentation, https://bluescarni.github.io/mppp/integer_benchmarks.html (accessed May 23, 2025).
[4] OpenMP Architecture Review Board, “OpenMP,” 维基百科, https://zh.wikipedia.org/wiki/OpenMP (accessed May 23, 2025).
[5] qq_44971458, "卷积计算加速方法——分块卷积," CSDN 博客, 2023 年 6 月 19 日, https://blog.csdn.net/qq_44971458/article/details/131282315