



南開大學
Nankai University

南 開 大 學

計 算 機 學 院

並行程序设计实验报告

Pthread OpenMP 多线程编程

侯嘉栋

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 5 月 29 日

摘要

本实验主要研究了并行程序设计在数值计算中的应用，特别是通过多线程编程优化复杂计算任务的性能。我们采用了多种并行化技术，包括 pthread 和 OpenMP，针对快速数论变换（NTT）和中国剩余定理（CRT）等数值计算进行了优化。实验过程中，我们通过不同的线程创建与回收机制、数据分块策略以及静态和动态调度方法，提升了计算效率。报告详细阐述了实验中使用的并行化策略，包括针对二进制分解、蝶形算法、块级并行化的优化策略，并对其性能改进效果进行了对比分析。实验结果表明，合理的并行优化能显著提升计算性能，但在小规模数据集或者线程创建过多时，也可能出现性能退化的问题。

关键字：Parallel OPenMP Pthread

目录

一、 概述	1
(一) 实验环境	1
(二) 仓库地址	1
(三) 第一节: Pthread 多线程编程	1
1. 动态创建-回收进程	2
2. 静态线程 + 信号量同步	4
3. 常驻线程 + 双 barrier	6
4. Radix-4 四分 NTT	7
5. CRT 多线程优化	12
6. DIF_DIT 的 NTT 实现	14
7. 线程数目对程序效果的影响	15
(四) 第二节:openmp 多线程编程	17
1. 朴素优化	17
2. Radix-4 多线程 NTT openmp 优化	18
3. CRT 的 openmp 优化	20
(五) 第三节: 使用 CRT 实现大模数	21
二、 总结	22

一、 概述

(一) 实验环境

```
1 NAME="openEuler"
2 VERSION="22.03 (LTS-SP4)"
3 ID="openEuler"
4 VERSION_ID="22.03"
5 PRETTY_NAME="openEuler 22.03 (LTS-SP4)"
6 ANSI_COLOR="0;31"
7 Linux 5.10.0-235.0.0.134.oe2203sp4.aarch64 aarch64 GNU/Linux
8 Static hostname: master_ubss1
9 Architecture: aarch64
10 CPU op-mode(s): 64-bit
11 Byte Order: Little Endian
12 CPU(s): 8
13 On-line CPU(s) list: 0-7
14 Vendor ID: HiSilicon
15 Model name: Kunpeng-920
16 Model: 0
17 Thread(s) per core: 1
18 Core(s) per cluster: 8
19 Socket(s): -
20 Cluster(s): 1
21 Stepping: 0x1
22 BogoMIPS: 200.00
23 NUMA node(s): 1
24 NUMA node0 CPU(s): 0-7
```

(二) 仓库地址

<https://github.com/iChubai/NKU-Parallel-Computing.git>

(三) 第一节: Pthread 多线程编程

NTT (数值变换) 算法中存在三层嵌套循环结构, 其中第二层和第三层均对整个输入数组进行遍历, 具有显著的并行计算潜力。具体而言, 第二层循环对应着 block 层的计算, 它负责在不同的块之间进行操作, 而第三层循环则对应着 butterfly 层, 其主要任务是对每对元素进行加减运算并应用旋转因子。这两个层级的计算操作具有独立性, 因此可以通过并行化策略显著提升计算效率。

针对这一点, 我们可以分别在 block 层和 butterfly 层应用多线程优化。在 block 层, 多线程可以通过同时处理多个块来加速数据的计算; 而在 butterfly 层, 在后续的实验中发现会存在线程爆炸的问题, 故后续未深入探索。通过对 block 层级的并行化优化, 我们能够有效地减少计算时间, 从而显著提升 NTT 算法的执行效率, 特别是在处理大规模数据时, 性能提升尤为明显。

Algorithm 1 NTT 伪代码

```

1: function NTT( $a, p, \text{roots}$ )
2:    $n \leftarrow \text{length}(a)$ 
3:   BitReversePermute( $a$ )
4:    $\text{len} \leftarrow 2$ 
5:   while  $\text{len} \leq n$  do ▷ stage 层
6:      $\text{half} \leftarrow \text{len}/2$ 
7:      $\text{step} \leftarrow n/\text{len}$ 
8:     for  $\text{start} \leftarrow 0$  to  $n - 1$  step  $\text{len}$  do ▷ block 层
9:        $\omega_{idx} \leftarrow 0$ 
10:      for  $i \leftarrow 0$  to  $\text{half} - 1$  do
11:         $u \leftarrow a[\text{start} + i]$ 
12:         $v \leftarrow a[\text{start} + i + \text{half}] \times \text{roots}[\omega_{idx}] \bmod p$ 
13:         $a[\text{start} + i] \leftarrow (u + v) \bmod p$  ▷ butterfly 层
14:         $a[\text{start} + i + \text{half}] \leftarrow (u - v + p) \bmod p$  ▷ butterfly 层
15:         $\omega_{idx} \leftarrow \omega_{idx} + \text{step}$ 
16:      end for
17:    end for
18:     $\text{len} \leftarrow \text{len} \times 2$ 
19:  end while
20: end function

```

1. 动态创建-回收进程

正如指导书介绍到：动态创建-回收进程是主线程才创建线程来进行并行计算，在这部分完成后，即销毁线程。在到达下一个并行部分时，再次重复创建线程——并行计算——销毁线程的步骤。优点是再没有并行计算需求时不会占用系统资源；缺点是有较大的线程创建和销毁开销。

对 block 层进行优化首先定义一个结构体用于传递参数：

结构体传参

```

1 typedef struct {
2     int *a; int len; int m; int wn; int p; int tid; int step; int lim;
3 } vl_arg_t;

```

之后将 block 层具体执行的代码抽出，封装成一个函数，供 pthread 多线程调用。

block 层封装函数

```

1 static void *vl_kernel(void *arg){
2     vl_arg_t *pa = (vl_arg_t*)arg;
3     int *a=pa->a, len=pa->len, m=pa->m, wn=pa->wn, p=pa->p, tid=pa->tid, step
        =pa->step, lim=pa->lim;
4     for(int i = tid*len; i < lim; i += step*len){
5         int w = 1;
6         for(int j=0; j<m; ++j){
7             int u=a[i+j], v=1LL*a[i+j+m]*w%p;
8             a[i+j]=(u+v)%p;
9             a[i+j+m]=(u-v+p)%p;

```

```

10         w=1LL*w*wn%p;}}
11     return NULL;}

```

之后 NTT 运行时每个循环都要创建进程，之后等待任务完成后销毁进程。

NTT 主函数

```

1 static void ntt_v1(int *a,int lim,int opt,int p){
2     int rev[lim]; get_rev(rev,lim);
3     for(int i=0;i<lim;++i) if(i<rev[i]) std::swap(a[i],a[rev[i]]);
4     for(int len=2;len<=lim;len<=1){
5         int m=len>>1;
6         int wn=qpow(3,(p-1)/len,p);
7         if(opt==1) wn=qpow(wn,p-2,p);
8         pthread_t th[V1_THREADS];
9         v1_arg_t param[V1_THREADS];
10        for(int t=0;t<V1_THREADS;++t){
11            param[t]={a,len,m,wn,p,t,V1_THREADS,lim};
12            pthread_create(&th[t],0,v1_kernel,&param[t]);
13        for(int t=0;t<V1_THREADS;++t) pthread_join(th[t],0);
14    if(opt==1){
15        int inv=qpow(lim,p-2,p);
16        for(int i=0;i<lim;++i) a[i]=1LL*a[i]*inv%p;}}

```

从线程生命周期与创建时机的角度分析,这个版本的代码每一级 len 都重新 pthread_create/join, 将产生 $2\log_2(lim \times T)$ 次的系统开销 (lim 是数组的大小, 即问题的规模, T 是系统进程数), 开销比较显著, 同时当系统调度的颗粒度远大于蝶形计算的颗粒度, 在大型的 NTT 里仍可能接受, 但在小规模 NTT, 或者是 len 在很接近 lim 时会存在浪费。从任务划分的角度来说, 此版本在前期 (len 小、段数多) 的时候, 负载均衡比较 OK, 但在后期 (len lim/2 或 = lim) 时, 段数 \leq 线程数, 会导致出现空闲线程, 极端时仅 1 段, 只有 tid=0 有事做, 其余线程 join 等待, 造成浪费。

表 1: 不同参数下的性能对比

参数 (n,p)	起始版本 (ms)	动态创建-回收进程多线程优化 (ms)	性能提升 (%)
$n = 4, p = 7340033$	0.00398	1.54406	$\approx -386\%$
$n = 131072, p = 7340033$	85.7989	53.272	$\approx 37\%$
$n = 131072, p = 104857601$	89.2863	50.2919	$\approx 43\%$
$n = 131072, p = 469762049$	92.1617	53.3789	$\approx 42\%$

使用 perf 测试其性能如下:

事件	数值	备注
cache-references	667,419,799	0.650% of all cache refs
cache-misses	4,335,472	
branch-instructions	<not supported>	
branch-misses	521,858	
time elapsed (s)	0.435393037	
user time (s)	0.592049000	
sys time (s)	0.198213000	

表 2: Perf 性能统计结果

对 butterfly 层优化

同理，我们只需要抽离出第三层循环的代码，将其封装为一个函数，进行类似的动态创建和回收即可。

NTT 主函数

```

1 static void ntt_v1_bfly(int *a, int lim, int opt, int p){
2     std::unique_ptr<int[]> rev(new int[lim]); get_rev(rev.get(), lim);
3     for(int i=0; i<lim; ++i) if(i<rev[i]) std::swap(a[i], a[rev[i]]);
4     pthread_t th[THREADS]; Arg pa[THREADS];
5     for(int len=2; len<=lim; len<<=1){
6         int m=len>>1, wn=qpow(3, (p-1)/len, p); if(opt==1) wn=qpow(wn, p-2, p);
7         int wnStep=qpow(wn, THREADS, p);
8         for(int i=0; i<lim; i+=len){
9             for(int t=0; t<THREADS; ++t){
10                 pa[t]={a, i, m, wn, wnStep, p, t};
11                 pthread_create(&th[t], 0, work, &pa[t]);
12                 for(int t=0; t<THREADS; ++t) pthread_join(th[t], 0);
13             if(opt==1){int inv=qpow(lim, p-2, p); for(int i=0; i<lim; ++i) a[i]=mulmod(a[i], inv, p);}}

```

相较于上个版本，这版 NTT 在 butterfly 层进行多线程，负载更均衡，但带来共享 cache-line 写的问题，cache 友好但尾段常出现空闲线程。同时本版本对每个块都 create/join THREADS 比较容易造成系统调用次数的爆炸。但这个版本代码有致命的问题，在实际运行中我发现其会创建非常多的线程，最终把操作系统的线程表撑爆了，只有在 n 非常小的时候（第一个测试样例）可以工作。

2. 静态线程 + 信号量同步

同样参考示例代码的思路，定义参数结构体和信号量，实现所有线程的一次性创建，消除 create 和 join 带来的循环开销，线程调度成本摊为 0

初始化

```

1 static void setup_v2_threads(){
2     static bool initd=false;
3     if(initd) return;
4     for(int i=0; i<V2_THREADS; ++i){ sem_init(&sem_start[i], 0, 0); sem_init(&
        sem_done[i], 0, 0); }

```

```

5   for(long long i=0;i<V2_THREADS;++i) pthread_create(&new pthread_t,0,
        v2_worker,(void*)i);
6   initied=true;}

```

线程生命周期和同步

```

1  static sem_t sem_start[T], sem_done[T];    // ← 私有 start / done
2  static void *v2_worker(void *tid_) {
3      int tid = (long long)tid_;
4      for (;;) {
5          sem_wait(&sem_start[tid]);          // 等待主线程发“开工”令
6          if (glob_len == 0) break;           // (未显式用到, 可后续扩展为退出
            信号)
7          ...                                // 干活
8          sem_post(&sem_done[tid]);           // 通知“完成”
9      }

```

主函数

```

1  static void ntt_v2(int *a,int lim,int opt,int p){
2      int rev[lim]; get_rev(rev,lim);
3      for(int i=0;i<lim;++i) if(i<rev[i]) std::swap(a[i],a[rev[i]]);
4      setup_v2_threads();
5      for(int len=2;len<=lim;len<<=1){
6          glob_a=a; glob_len=len; glob_m=len>>1;
7          glob_wn=qpow(3,(p-1)/len,p); if(opt==1) glob_wn=qpow(glob_wn,p-2,p);
8          glob_p=p; glob_lim=lim;
9          for(int t=0;t<V2_THREADS;++t) sem_post(&sem_start[t]);
10         for(int t=0;t<V2_THREADS;++t) sem_wait(&sem_done[t]);
11         if(opt==1){
12             int inv=qpow(lim,p-2,p);
13             for(int i=0;i<lim;++i) a[i]=1LL*a[i]*inv%p;}

```

任务划分的角度来说, 比起动态创建的方案, 这个版本缓存友好, 且进程之间没有交叉, 在 len 比较小的阶段, 段数远大于线程数, 负载均衡比较好, 在 len 比较大的阶段 (段数小于线程数时), 会出现闲线程, 但由于两级本身的计算量比较大, 空线程数小于总线程的一半, 所以对总时长的影响比较小。从计算阶段参数的广播的角度来说, 主线程先写全局只读变量, 再通过 `sem_post`, 由于 semaphore 是 release 操作, 其他线程 acquire 后即可看到最新值。

表 3: 不同参数下的性能对比

参数 (n, p)	起始版本 (ms)	静态线程 + 信号量同步 (ms)	性能提升 (%)
$n = 4, p = 7340033$	0.00398	0.602691	$\approx -150\%$
$n = 131072, p = 7340033$	85.7989	43.2521	$\approx 49\%$
$n = 131072, p = 104857601$	89.2863	43.9929	$\approx 50\%$
$n = 131072, p = 469762049$	92.1617	46.4419	$\approx 49\%$

事件	数值	备注
cache-references	667,019,768	0.640% of all cache refs
cache-misses	4,269,343	
branch-instructions	<not supported>	
branch-misses	534,934	
time elapsed (s)	0.642002048	
user time (s)	0.596596000	
sys time (s)	0.094741000	

表 4: Perf 性能统计结果

3. 常驻线程 + 双 barrier

参考指导书的做法, 我们也可以采取常驻线程 + 双 barrier 的方式.

双 barrier

```

1 static void *v3_kernel(void *arg){
2     int tid=(long long) arg;
3     for(;;){
4         pthread_barrier_wait(&bar1);
5         if(bar_len==0) break;
6         for(int i=tid*bar_len;i<bar_lim;i+=V3_THREADS*bar_len){
7             int w=1;
8             for(int j=0;j<bar_m;++j){
9                 int u=bar_a[i+j], v=1LL*bar_a[i+j+bar_m]*w%bar_p;
10                bar_a[i+j]=(u+v)%bar_p;
11                bar_a[i+j+bar_m]=(u-v+bar_p)%bar_p;
12                w=1LL*w*bar_wn%bar_p;}}
13        pthread_barrier_wait(&bar2);}
14    return NULL;}
15 static void setup_v3_threads(){
16     static bool initd=false;
17     if(initd) return;
18     pthread_barrier_init(&bar1,0,V3_THREADS+1);
19     pthread_barrier_init(&bar2,0,V3_THREADS+1);
20     for(long long i=0;i<V3_THREADS;++i) pthread_create(&bar_th[i],0,v3_kernel
21         ,(void*)i);
22     initd=true;}
23 static void ntt_v3(int *a,int lim,int opt,int p){
24     std::unique_ptr<int[]> rev(new int[lim]);
25     get_rev(rev.get(),lim);
26     for(int i=0;i<lim;++i) if(i<rev[i]) std::swap(a[i],a[rev[i]]);
27     setup_v3_threads();
28     for(int len=2;len<=lim;len<<=1){
29         bar_a=a; bar_len=len; bar_m=len>>1;
30         bar_wn=qpow(3,(p-1)/len,p); if(opt===-1) bar_wn=qpow(bar_wn,p-2,p);
31         bar_p=p; bar_lim=lim; bar_opt=opt;
32         pthread_barrier_wait(&bar1);

```



```

32     pthread_barrier_wait(&bar2);}
33     if(opt==-1){
34         int inv=qpow(lim,p-2,p);
35         for(int i=0;i<lim;++i) a[i]=1LL*a[i]*inv%p;}}

```

barrier1 充当“开工令”，主线程更新全局变量后等待.barrier2 充当“完工令”，所有工线程都算完后释放主线程，进入下一级。因此每个阶段只有一次全员同步，代价比较轻，在任务划分方面，其实现了无写交叉，继承了之前版本的内存访问优势，同时负载均衡的分析同理，barrier 写法更短、更不易写错。通过实验，我们测得的结果如下：

表 5: 不同参数下的性能对比

参数 (n, p)	起始版本 (ms)	常驻线程 + 双 barrier (ms)	性能提升 (%)
$n = 4, p = 7340033$	0.00398	1.02399	$\approx -256\%$
$n = 131072, p = 7340033$	85.7989	48.9863	$\approx 42\%$
$n = 131072, p = 104857601$	89.2863	48.3515	$\approx 45\%$
$n = 131072, p = 469762049$	92.1617	46.4419	$\approx 49\%$

事件	数值	备注
cache-references	666,667,314	0.636% of all cache refs
cache-misses	4,243,048	
branch-instructions	<not supported>	
branch-misses	543,317	
time elapsed (s)	0.468748502	
user time (s)	0.618170000	
sys time (s)	0.145768000	

表 6: Perf 性能统计结果 for ./ntt

4. Radix-4 四分 NTT

之前我们一直对 radix-2 [1] 的 NTT 版本进行优化，现在我们考虑 radix-4 版本 [2]

设序列 $x[n]$ 的长度为 $N=4^M$ ， M 为整数。如果不满足这个条件，可以通过补零，使之达到这个要求。我们通过抽取将 $x[n]$ 分为四个长度为 $\frac{N}{4}$ 的子序列如下：

$$x_1[n] = x[4n] \quad x_2[n] = x[4n + 1] \quad x_3[n] = x[4n + 2] \quad x_4[n] = x[4n + 3]$$

则 $x[n]$ 的 DFT 可以表示为:

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

$$= \sum_{n=0}^{\frac{N}{4}-1} (x[4n] W_N^{4nk} + x[4n+1] W_N^{(4n+1)k} + x[4n+2] W_N^{(4n+2)k} + x[4n+3] W_N^{(4n+3)k}) \quad (2)$$

$$= \sum_{n=0}^{\frac{N}{4}-1} \left(x[4n] W_{\frac{N}{4}}^{nk} + x[4n+1] W_{\frac{N}{4}}^{nk} W_N^k + x[4n+2] W_{\frac{N}{4}}^{nk} W_N^{2k} + x[4n+3] W_{\frac{N}{4}}^{nk} W_N^{3k} \right) \quad (3)$$

$$= X_1(k) + W_N^k X_2(k) + W_N^{2k} X_3(k) + W_N^{3k} X_4(k), \quad (4)$$

其中

$$X_i(k) = \sum_{n=0}^{\frac{N}{4}-1} x_i[n] W_{\frac{N}{4}}^{nk}, \quad i = 1, 2, 3, 4,$$

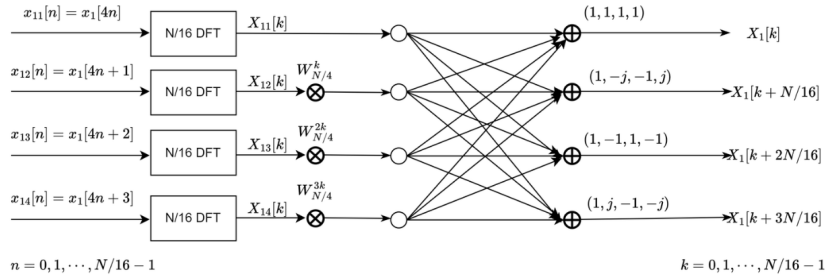
且对 $k = 0, 1, \dots, \frac{N}{4} - 1$ 之外的下标, 有

$$X(k + \frac{N}{4}) = X_1(k) - j W_N^k X_2(k) - W_N^{2k} X_3(k) + j W_N^{3k} X_4(k), \quad (5)$$

$$X(k + \frac{2N}{4}) = X_1(k) - W_N^k X_2(k) + W_N^{2k} X_3(k) - W_N^{3k} X_4(k), \quad (6)$$

$$X(k + \frac{3N}{4}) = X_1(k) + j W_N^k X_2(k) - W_N^{2k} X_3(k) - j W_N^{3k} X_4(k). \quad (7)$$

对于序列 $x_1[n], x_2[n], x_3[n], x_4[n]$, 我们可以继续用上述“分而治之”(divide and conquer) 的方法将每个序列分为四个长度为 $\frac{N}{16}$ 的子序列。下面给出 $X_1[n]$ 划分的示意图:



我们可以继续用上述的方法将每个子序列继续划分, 直到最后序列长度为 4。4 点的 DFT 计算如下:

$$\begin{bmatrix} X[0] \\ X[1] \\ X[2] \\ X[3] \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & -j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & j \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ x[3] \end{bmatrix}.$$

Algorithm 2 Radix-4 并行数论变换 NTT_RADIX4

Input: 多项式系数数组 $A[0 \dots N-1]$, 模数 p , 原根 g , 方向标志 $invert \in \{0, 1\}$, 线程数 T

Output: A 原地变换为 $\text{NTT}(A)$ (当 $invert = 0$) 或 $\text{NTT}^{-1}(A)$ (当 $invert = 1$)

1: **if** $invert = 1$ **then**

2: $g \leftarrow g^{p-2} \pmod{p}$

▷ 取原根的逆

```

3: end if
4: DIGITREVERSE4( $A, N, T$ ) ▷ 基 -4 位反转
5: for  $len \leftarrow 4$  to  $N$  step  $len \leftarrow 4 \times len$  do
6:    $m \leftarrow len/4$ 
7:    $wn \leftarrow g^{\frac{p-1}{len}} \pmod{p}$ 
8:    $J \leftarrow wn^m \pmod{p}$  ▷  $J = \sqrt{-1} \pmod{p}$ 
9:   预生成根表  $W[0 \dots m-1]$  并行安全
       $W[0] \leftarrow 1; W[j] \leftarrow W[j-1] \cdot wn \pmod{p}$ 
      ( $blk = 0$  to  $N-1$  step  $len$ , 使用  $T$  线程)
10:  for  $j = 0$  to  $m-1$  do
11:     $w_1 \leftarrow W[j]; w_2 \leftarrow w_1^2 \pmod{p}; w_3 \leftarrow w_2 \cdot w_1 \pmod{p}$ 
      // 裁取并乘相位
12:     $A_0 \leftarrow A[blk+j]$ 
13:     $A_1 \leftarrow A[blk+j+m] \cdot w_1 \pmod{p}$ 
14:     $A_2 \leftarrow A[blk+j+2m] \cdot w_2 \pmod{p}$ 
15:     $A_3 \leftarrow A[blk+j+3m] \cdot w_3 \pmod{p}$ 
      // Radix-4 蝶形
16:     $T_0 \leftarrow A_0 + A_2 \pmod{p}; T_1 \leftarrow A_0 - A_2 \pmod{p}$ 
17:     $T_2 \leftarrow A_1 + A_3 \pmod{p}; T_3 \leftarrow (A_1 - A_3) \cdot J \pmod{p}$ 
18:     $A[blk+j] \leftarrow T_0 + T_2 \pmod{p}$ 
19:     $A[blk+j+m] \leftarrow T_1 + T_3 \pmod{p}$ 
20:     $A[blk+j+2m] \leftarrow T_0 - T_2 \pmod{p}$ 
21:     $A[blk+j+3m] \leftarrow T_1 - T_3 \pmod{p}$ 
22:  end for
23: end for
24: if  $invert = 1$  then
25:    $invN \leftarrow N^{p-2} \pmod{p}$   $i = 0$  to  $N-1$ 
26:    $A[i] \leftarrow A[i] \cdot invN \pmod{p}$ 
27: end if

```

Algorithm 3 DIGITREVERSE4 — 基 -4 位反转

```

1: function DIGITREVERSE4( $A, N, T$ )
2:    $k \leftarrow \log_4 N$  ▷ 总 2-bit 位数  $i = 0$  to  $N-1$  使用  $T$  线程
3:    $rev \leftarrow 0, tmp \leftarrow i$ 
4:   for  $d = 0$  to  $k-1$  do
5:      $rev \leftarrow (rev \ll 2) \mid (tmp \wedge 3)$ 
6:      $tmp \leftarrow tmp \gg 2$ 
7:   end for
8:   if  $i < rev$  then
9:     交换  $A[i] \leftrightarrow A[rev]$ 
10:  end if
11: end function

```

```

1 struct LoopArg{
2     long long beg,end,step;
3     const std::function<void(long long)> *body;    // 安全的堆指针};
4 void* loop_worker(void* arg){
5     auto* A = static_cast<LoopArg*>(arg);
6     for(long long i=A->beg;i<A->end;i+=A->step) (*(A->body))(i);
7     return nullptr;}
8 template<typename F>
9 inline void parallel_for(long long beg,long long end,long long step,
10                          const F& func,int THREADS)
11 {   if(end - beg <= step * THREADS){           // 小任务串行
12     for(long long i=beg;i<end;i+=step) func(i);
13     return;}
14     auto* body = new std::function<void(long long)>(func);    // 堆上
15     long long chunk = ((end - beg) + THREADS - 1)/THREADS;
16     std::vector<pthread_t> threads(THREADS);
17     std::vector<LoopArg> args(THREADS);
18     for(int t=0;t<THREADS;++t){
19         long long l = beg + t*chunk;
20         long long r = std::min<long long>(end, l + chunk);
21         args[t] = { l, r, step, body };
22         pthread_create(&threads[t], nullptr, loop_worker, &args[t]);
23     for(auto& th : threads) pthread_join(th, nullptr);
24     delete body;                                           // 回收}
25 void digrev4(int *a,int n,int T){
26     int pairs = __builtin_ctz(n)>>1;
27     parallel_for(0,n,1,[&](long long idx){
28         int i=int(idx), rev=0, tmp=i;
29         for(int j=0;j<pairs;j++){ rev=(rev<<2)|(tmp&3); tmp>>=2; }
30         if(i<rev) std::swap(a[i],a[rev]);
31     },T);}

```

这次我们采用堆指针传递函数指针，在较小的任务上采用串行方法，在较大的任务上并行，需要注意的是四分的 NTT 需要修改计算 rev 数组的方式，具体细节可以查阅相关资料。根据上述公式可以写出函数的主体部分：

Radix-4 四分 NTT 函数主体

```

1 void ntt_rad4(int *a,int n,bool inv,int p,int T){
2     digrev4(a,n,T);
3     for(int len=4; len<=n; len<<=2){
4         int m=len>>2;int wn=qpow(3,(p-1)/len,p);
5         if(inv) wn=qpow(wn,p-2,p);int J=qpow(wn,m,p);
6         std::vector<int> wtab(m); wtab[0]=1;
7         for(int j=1;j<m;j++) wtab[j]=1LL*wtab[j-1]*wn%p;
8         parallel_for(0,n,len,[&](long long blk){
9             for(int j=0;j<m;j++){
10                 int w1=wtab[j],w2=1LL*w1*w1%p,w3=1LL*w2*w1%p;
11                 int A=a[blk+j];int B=1LL*a[blk+j+m]*w1%p;

```

```

12  int C=1LL*a[blk+j+2*m]*w2%p; int D=1LL*a[blk+j+3*m]*w3%p;
13  int T0=(A+C)%p, T1=(A+p-C)%p; int T2=(B+D)%p, T3=1LL*(B+p-D)*J%p;
14  a[blk+j]=(T0+T2)%p; a[blk+j+m]=(T1+T3)%p;
15  a[blk+j+2*m]=(T0+p-T2)%p;
16  a[blk+j+3*m]=(T1+p-T3)%p; } } ,T); if(inv){ int invN=qpow(n,p-2,p);
17  parallel_for(0,n,1,[=(long long i){ a[i]=1LL*a[i]*invN%p; },T); }}

```

内部自写的 `parallel_for` 先把迭代区间按整块长度 $\text{chunk} = \text{ceil}((\text{end}-\text{beg})/T)$ 切段, 线程 t 只处理 $[\text{beg}+t \cdot \text{chunk}, \min(\text{end}, \dots))$ 。这种静态划分在 NTT 场景有两点优势:

1. 每线程访问的数组子区间连续, 硬件预取命中高, 跨 NUMA 情况下页映射稳定;
2. 迭代空间总长远大于线程数, `chunk` 差 1 段不足 0.01 %, 负载差异可忽略

在蝴蝶并行阶段, 外层 `blk += len` 循环被交给 `parallel_for`:

每线程一次性获得若干完整块, 块大小 $\geq \text{len}$ (最小 2 KB)。每线程一次性获得若干完整块, 块大小 $\leq \text{len}$ (最小 2 KB)。

当进入 radix-4 路径时, 一个块容纳 4 个系数、算访比更高, 并行度仍由块数 $= \frac{N}{\text{len}}$ 决定; 前期层线程满载, 后期层块数爆增, 负载自动均匀。若 $\log N$ 为奇数, 本实现整条路径回退到 radix-2, 避免“radix-4 需先补一层 2-叉再 4-叉”那种折返访问, 从而少一次 create-join, 也防止最后一级块尺寸过小造成线程空转。

每次 `parallel_for` 把待执行 `lambda` 包装进堆对象, 线程获得 `const std::function*`, 保证在 join 之前对象仍存活。回收 `delete body` 发生在所有线程 join 之后, 完全避免“double free”风险

总之, 这版并行优化把并行重心放在块级负载均衡和缓存友好上, 而把同步问题交给 Linux 调度器处理

表 7: 不同参数下的性能对比

参数 (n, p)	起始版本 (ms)	Radix-4 四分 NTT (ms)	性能提升 (%)
$n = 4, p = 7340033$	0.00398	0.02279	$\approx -472\%$
$n = 131072, p = 7340033$	85.7989	61.2307	$\approx 28\%$
$n = 131072, p = 104857601$	89.2863	53.684	$\approx 39\%$
$n = 131072, p = 469762049$	92.1617	69.5587	$\approx 24\%$

perf 的性能测试如下:

事件	数值	备注
cache-references	562,922,809	0.737% of all cache refs
cache-misses	4,149,496	
branch-instructions	<not supported>	
branch-misses	616,910	
time elapsed (s)	0.363778165	
user time (s)	0.495120000	
sys time (s)	0.201002000	

表 8: Perf 性能统计结果 for `./ntt`

5. CRT 多线程优化

首先我们先了解 CRT 是什么：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, \dots, n_k 两两互质)

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

我们将按如下的方式计算问题的解：

1. 计算所有模数的积 n ;
2. 对于第 i 个方程
 - (a) 计算 $m_i = \frac{n}{n_i}$
 - (b) 计算 m_i 在模 n_i 意义下的逆元 m_i^{-1}
 - (c) 计算 $c_i = m_i m_i^{-1}$ (不对 n_i 取模)
3. 方程组在模 n 意义下的唯一解为: $x = \sum_{i=1}^k a_i c_i \pmod{n}$

证明：

我们需要证明算法求得的 x 对任意 $i = 1, 2, \dots, k$ 都满足

$$x \equiv a_i \pmod{n_i}.$$

当 $i \neq j$ 时, 有 $m_j \equiv 0 \pmod{n_i}$, 故

$$c_j \equiv m_j \equiv 0 \pmod{n_i}.$$

又因

$$c_i \equiv m_i (m_i^{-1} \pmod{n_i}) \equiv 1 \pmod{n_i},$$

于是

$$\begin{aligned} x &\equiv \sum_{j=1}^k a_j c_j \pmod{n_i} \\ &\equiv a_i c_i \pmod{n_i} \\ &\equiv a_i m_i (m_i^{-1} \pmod{n_i}) \pmod{n_i} \\ &\equiv a_i \pmod{n_i}. \end{aligned}$$

因此, 对任意 $i = 1, 2, \dots, k$, 算法输出的 x 都满足 $x \equiv a_i \pmod{n_i}$, 从而证明了该同余方程组求解算法的正确性。

由于我们没有对输入的 a_i 施加任何特殊限制, 任意一组 $\{a_i\}_{i=1}^k$ 都对应唯一的解 x 。

另一方面, 若 $x \neq y$, 则必存在某个 i 使得 $x \not\equiv y \pmod{n_i}$ 。

根据证明中的构造, 我们可以得到 **k 模公式**。

$$x \equiv \sum_{j=1}^k a_j m_j (m_j^{-1} \pmod{n_j}) \pmod{n}$$

Algorithm 4 三模数中国剩余定理 (CRT) 合并——多项式乘法结果**Input:** 三个模数 m_1, m_2, m_3 , 对应的 NTT 结果 r_1, r_2, r_3 , 目标模数 mod **Output:** 合并后的结果 x (模 mod)

```

1: function CRTTHREEMOD( $r_1, r_2, r_3, m_1, m_2, m_3, mod$ )
2:   初始化结果  $x \leftarrow 0$ 
3:   for  $k = 1$  to 3 do                                     ▷ 对每个模数进行处理
4:     计算  $M_k \leftarrow \prod_{j \neq k} m_j \bmod mod$                 ▷ 其他模数的乘积
5:     计算  $M'_k \leftarrow M_k \bmod m_k$                         ▷ 取模以计算逆元
6:     计算  $b_k \leftarrow \text{modInverse}(M'_k, m_k)$                 ▷ 模逆元,  $M'_k \cdot b_k \equiv 1 \bmod m_k$ 
7:     计算  $term \leftarrow (r_k \cdot b_k \cdot M_k) \bmod mod$       ▷ 当前模数的贡献
8:      $x \leftarrow (x + term) \bmod mod$                         ▷ 累加结果
9:   end for
10:  return  $x$ 
11: end function
12: function MODINVERSE( $a, m$ )                                ▷ 计算模逆元
13:   使用扩展欧几里得算法或费马小定理计算  $a^{-1} \bmod m$ 
14:  return  $a^{-1}$ 
15: end function
16: 对于多项式乘法结果的每个系数  $i$ :
17:   运行 NTT 计算, 得到三个模数下的结果  $r_1[i], r_2[i], r_3[i]$ 
18:   调用 CRTThreeMod( $r_1[i], r_2[i], r_3[i], m_1, m_2, m_3, mod$ )
19:   将结果存储到输出数组
20: return 合并后的多项式系数数组

```

CRT 主要的并行化 [3] 机会在于每个模数 NTT 的计算是相互独立的, 它们可以被分配到不同的线程中并发执行。这是最显著的并行点, 因为 NTT 本身是计算密集型操作。

k 模合并

```

1 inline void poly_multiply_crt(u64 *a, u64 *b, u64 *ab, u64 n, u64 p) {
2   u64 n_expanded = expand_n(2 * n - 1);
3   u64 **ab_crt = new u64 *[CRT_NUMS];
4   for (u64 i = 0; i < CRT_NUMS; i++) {
5     ab_crt[i] = new u64[n_expanded]{};
6   }
7   std::vector<std::thread> threads;
8   threads.reserve(CRT_NUMS);
9   for (u64 i = 0; i < CRT_NUMS; i++) {
10    threads.emplace_back(poly_multiply_ntt, a, b, ab_crt[i], n, CRT_MODS[i]);
11  }
12  for (u64 i = 0; i < CRT_NUMS; i++) {
13    if (threads[i].joinable()) {
14      threads[i].join();
15    }
16  }
17  u128 *ab_u128 = new u128[n_expanded];
18  for (u64 i = 0; i < n_expanded; ++i) ab_u128[i] = ab_crt[0][i];
19  CRT_combine(ab_u128, ab_crt, n_expanded);
20  for (u64 i = 0; i < n_expanded; ++i) ab[i] = ab_u128[i] % p;
21  delete [] ab_u128;

```

```

18     for (u64 i = 0; i < CRT_NUMS; ++i) delete[] ab_crt[i];
19     delete[] ab_crt;
20 }

```

我们创建了三个线程，分别使用三个不同的小模数 (998244353, 1004535809, 469762049)，得到三个 NTT 后的结果数组，之后按照公式进行三模数的合并。

三模合并，frame

```

1 inline void CRT_combine(u128 *ab, u64 **ab_crt, u64 n) {
2     u128 m = CRT_MODS[0];
3     for (u64 i = 1; i < CRT_NUMS; ++i) {
4         ModGeneric<u64> mod_op(CRT_MODS[i]);
5         u128 inv = mod_op.inv(m % CRT_MODS[i]);
6         for (u64 j = 0; j < n; j++) {
7             u128 x = ab[j];
8             u64 t = mod_op.sub(ab_crt[i][j], x % CRT_MODS[i]);
9             ab[j] = x + m * mod_op.mul(t, inv);
10        m *= CRT_MODS[i];
11    }
12 }

```

通过 `std::vector<std::thread>` 容器存储线程对象，使用 `emplace_back` 创建并启动线程，最后通过 `join()` 等待所有 NTT 线程执行完毕。这种方式确保了在进入 CRT 合并阶段前，所有并行的 NTT 计算均已完成。并行粒度是整个 `poly_multiply_ntt` 函数的执行。由于 `CRT_NUMS` 通常较小（如 3 个），且假设对于相同的输入 `n`，不同 `CRT_MODS[i]` 下的 NTT 计算时间相近，这种任务分配自然地实现了较好的负载均衡。上述代码的运行结果是：但 CRT 的方法比价适合大

表 9: 不同参数下的性能对比

参数 (n, p)	起始版本 (ms)	CRT 多线程优化	性能提升 (%)
$n = 4, p = 7340033$	0.00398	7.27632	$\approx -1827\%$
$n = 131072, p = 7340033$	85.7989	131.281	$\approx -53\%$
$n = 131072, p = 104857601$	89.2863	136.572	$\approx -52\%$
$n = 131072, p = 469762049$	92.1617	132.384	$\approx -43\%$

模数，我们进行测试可以看到性能都发生了严重的退化，尤其是在 n 为 4 这一非常小的模数上。

6. DIF_DIT 的 NTT 实现

承接上一个实验，我采用 Pthreads 对数论变换 (NTT) 的 DIF (频率抽取) 和 DIT (时间抽取) 算法进行并行优化。主要针对以下部分进行优化：

1. DIF/DIT 蝶形运算阶段：每个阶段的蝶形运算被划分为若干计算块 (blocks)。这些块被分配给固定数量的 Pthreads (NUM_THREADS) 进行并行处理。每个线程负责处理一部分蝶形运算，通过 `start_block_idx` 和 `end_block_idx` 控制其工作范围。
2. 点积运算：在多项式乘法的频域表示中，系数的点对点相乘是独立进行的。这一步骤同样通过将整个数组划分为段，分配给多个线程并行计算，从而加速处理。
3. DIT 末尾缩放因子 (Scaling Factor) 应用：DIT 逆变换的最后一步是将所有系数乘以模逆元 `invl`。这一操作也是元素独立的，因此可以有效地并行化，每个线程负责数组的一个子

段。

实现的代码采用了动态创建-回收进程的方式，与之前实现比较相似。在此不多赘述。此处的 Baseline 选择了上次实验实现的串行 DIF_DIT 代码。

表 10: pthread 版相对串行版的性能提升（负号表示变慢）

参数 (n, p)	串行 (ms)	pthread (ms)	性能提升 (%)
$n = 4, p = 7,340,033$	0.00840	1.62935	-1.93×10^4
$n = 131\,072, p = 7,340,033$	163.075	167.184	-2.52
$n = 131\,072, p = 104\,857\,601$	132.675	166.786	-25.7
$n = 131\,072, p = 469\,762\,049$	132.873	165.698	-24.7

可以发现此处频繁的创作销毁进程反而造成了性能的损失。

7. 线程数目对程序效果的影响

我选用了动态创建-回收进程，静态线程 + 信号量同步，常驻线程 + 双 barrier，(以下分别以 main_pthread_v1、main_pthread_v2、main_pthread_v3 代称) 这三个版本的代码来进行本次的实验。

对如下几个指标进行测试：

- 效率：即加速比除以线程数
- 延迟：即代码运行的时间
- 加速比
- 平均延迟

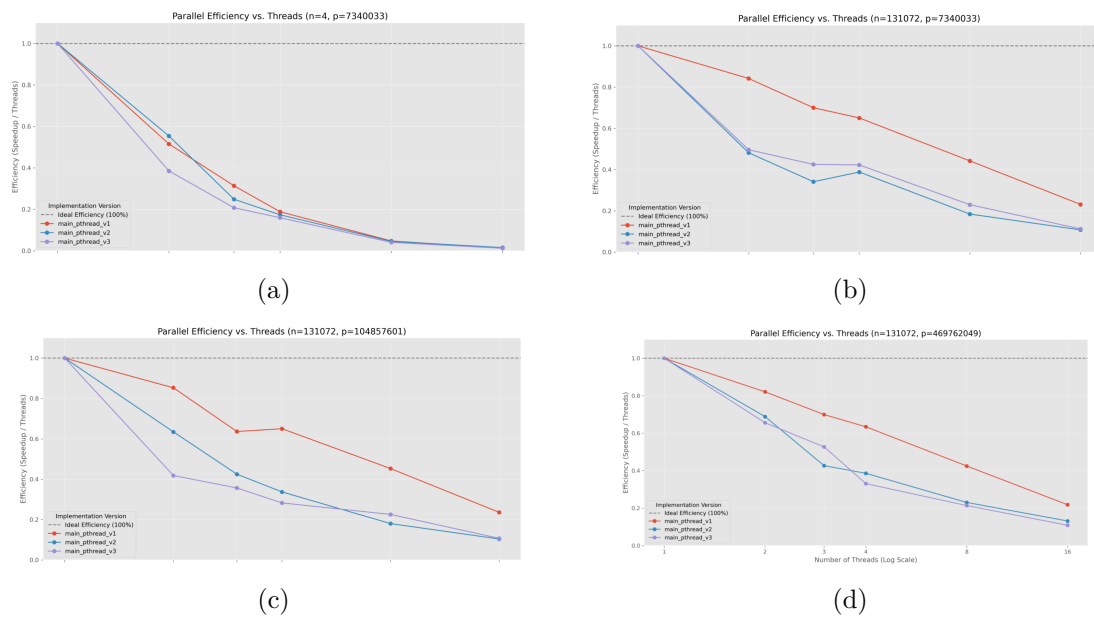


图 1: 并行计算的效率

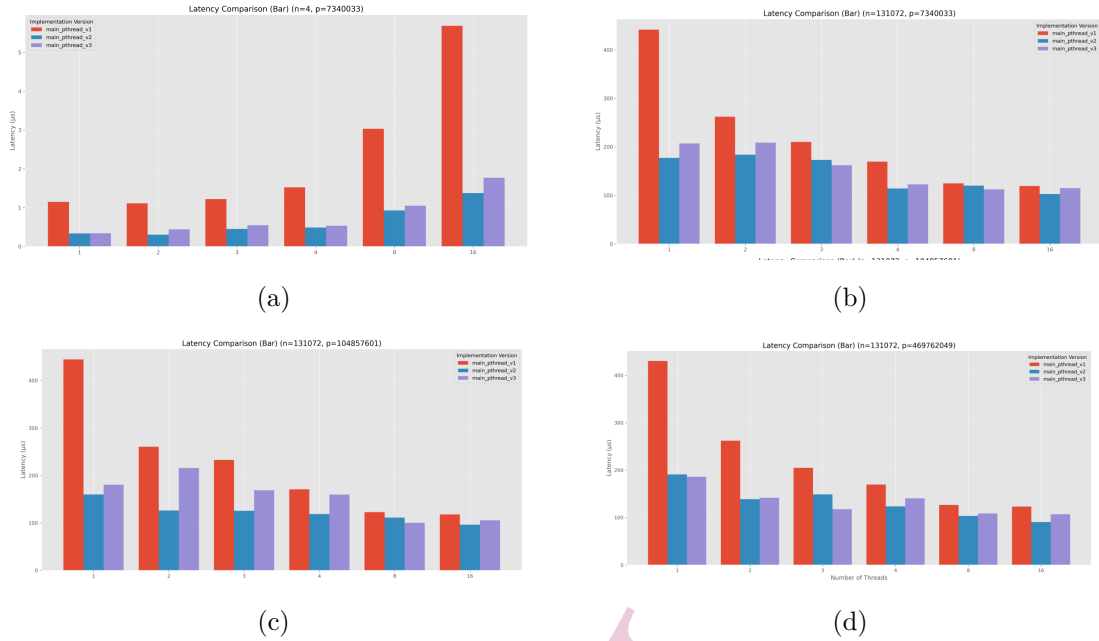


图 2: 延迟比较

对于小规模问题 ($n = 4$, $p = 7340033$): 折线图清晰显示加线程数通常会导致延迟增加而不是减少。这是因为并行化带来的开销 (线程创建、同步、管理) 远远超过了并行计算本身能够节省的时间。main_thread_v2 和 main_thread_v3 (使用常驻线程池的思路, 如信号量或屏障) 的单线程延迟 ($0.338 \mu s$ 和 $0.341 \mu s$) 显著低于 main_thread_v1 ($1.148 \mu s$)。这表明对于小任务, 动态创建和销毁线程 (v1) 的开销较大。即使是 v2 和 v3, 随着线程数从 1 增加到 16, 延迟也呈现上升趋势 (例如 v2: $0.338 \rightarrow 1.376 \mu s$; v3: $0.341 \rightarrow 1.771 \mu s$)。可见对于 $n=4$ 这样的小问题, 并行化是不划算的。单线程执行 (尤其是采用 v2 或 v3 的低开销单线程模式) 是最优选择。

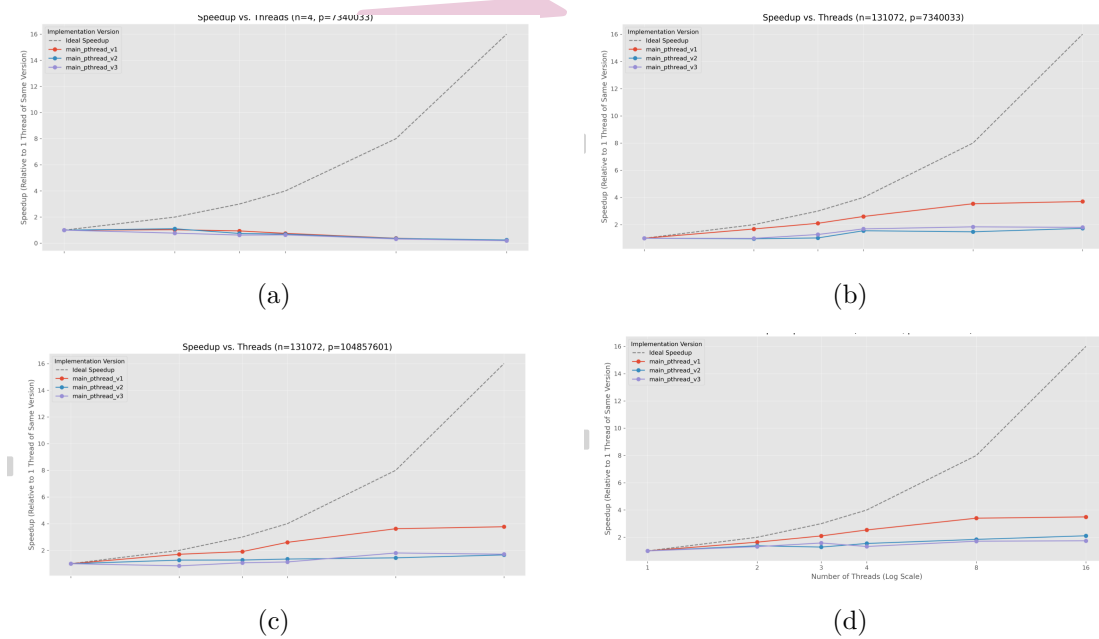


图 3: 加速比

对于大规模问题 ($n = 131072$, 三种不同模数): 并行化优势开始显现。三种实现均随线程数增加而显著缩短延迟, 且降幅差异明显。v2 在 16 线程下达到最低延迟 (约 $90 \mu s$, $p = 469762049$), 较其单线程基线提升逾 52%, 并显著优于 v1/v3。v1 虽绝对性能最差, 但相对降幅最大, 显示可加速空间更大。对于加速比与并行效率, 以各自单线程性能为基准计算, v1/v2/v3 在 16 线程下的加速比分别约为 3.5、2.1、1.7, 对应并行效率约为 0.22、0.13、0.11。v1 的相对指标占优, 源于其单线程固定开销较高; 然而该优势并未转化为更低的实际延迟。总体而言, 三种实现的效率均随线程数递减, 说明同步、内存带宽与缓存竞争已成为主要瓶颈。在实现机制上比较, v2 采用静态线程池结合信号量/条件变量的生产者-消费者模式, 在绝对性能上表现最佳, 适合对延迟极敏感的场景。v3 通过屏障同步, 其单线程性能接近 v2, 但在多线程场景下因同步点等待开销而略逊。v1 每次调用动态创建/销毁线程, 虽能获得较高“表观”加速比, 却难以弥补较大的线程管理开销。

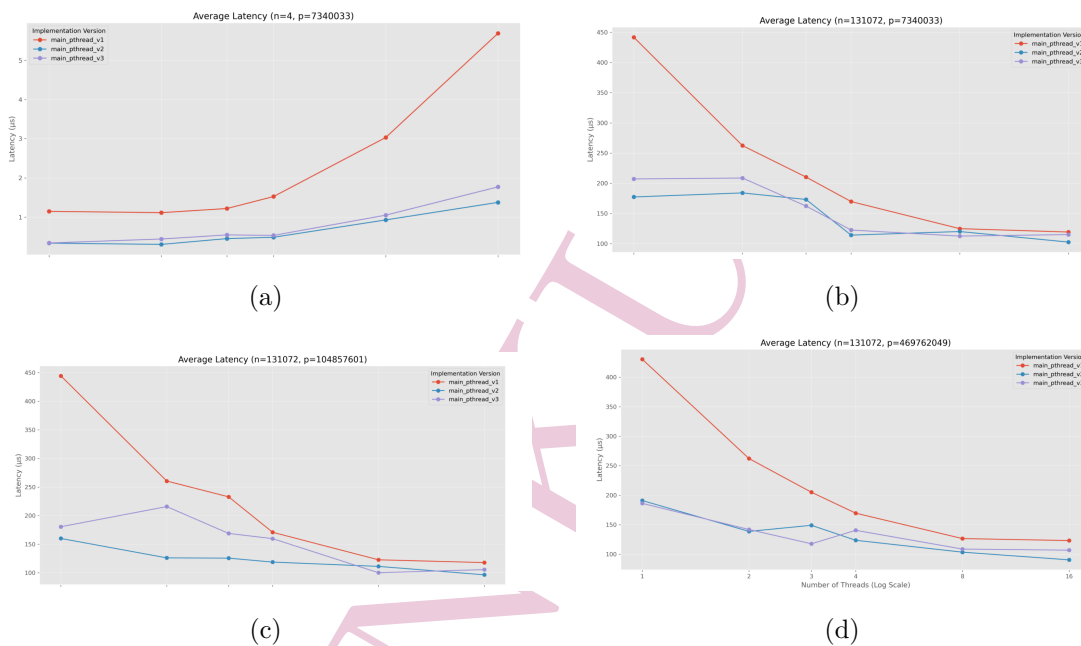


图 4: 平均延迟

(四) 第二节:openmp 多线程编程

1. 朴素优化

由于 openmp 是一个封装的很好, 很简单使用的库, 所以朴素优化的想法非常直接, 就是并行处理所有的循环。在算 rev 数组, 以及 NTT 的第二层循环前设置多线程并行 [4]。

朴素并行优化 NTT

```

1 #pragma omp parallel for schedule(static)
2 for(int i=0;i<lim;++i)
3     if(i<rev[i]) std::swap(a[i],a[rev[i]]);
4 #pragma omp parallel for schedule(static)
5     for(int i=0;i<lim;i+=len)
6 #pragma omp parallel for schedule(static)
7 for(int i=0;i<lim;++i) a[i]=1LL*a[i]*inv%p;

```

表 11: 不同参数下的性能对比

参数 (n, p)	起始版本 (ms)	朴素 openmp 优化 NTT(ms)	性能提升 (%)
$n = 4, p = 7340033$	0.00398	0.700561	$\approx -1.75 \times 10^4\%$
$n = 131072, p = 7340033$	85.7989	27.376	$\approx 68.1\%$
$n = 131072, p = 104857601$	89.2863	26.5424	$\approx 70.3\%$
$n = 131072, p = 469762049$	92.1617	29.9133	$\approx 67.5\%$

虽然想法很直接,但是效果很好,可见 openmp 底层的强大。有一个小 trick 就是第二层循环并行使用

```
1 #pragma omp parallel for collapse(2) schedule(static)
```

collapse(2) 表示将紧接着的两层完全嵌套的 for 循环视为一个大循环. 大迭代空间更好地负载均衡, 尤其当内层循环迭代数很少或不均匀时, 同时减少调度开销

perf 测试其性能如下:

事件	数值	备注
cache-references	814,609,060	0.519% of all cache refs
cache-misses	4,224,188	
branch-instructions	<not supported>	
branch-misses	518,761	
time elapsed (s)	0.334099658	
user time (s)	0.604462000	
sys time (s)	0.104054000	

表 12: Perf 性能统计结果

2. Radix-4 多线程 NTT openmp 优化

在上一节的基础上对 NTT 算法代码中的 for 循环多线程并行

openmp 优化 Radix-4

```
1 void ntt_rad4(int *a, int n, bool inv, int p){
2   digrev4(a, n);
3   for(int len=4; len<=n; len<=<=2){
4     int m=len>>2;
5     int wn=qpow(3, (p-1)/len, p);
6     if(inv) wn=qpow(wn, p-2, p);
7     int J=qpow(wn, m, p); // √-1
8     std::vector<int> wtab(m); wtab[0]=1;
9     for(int j=1; j<m; j++) wtab[j]=1LL*wtab[j-1]*wn%p;
10    #pragma omp parallel for schedule(static)
11    for(int blk=0; blk<n; blk+=len){
12      for(int j=0; j<m; j++){
13        int w1=wtab[j];
14        int w2=1LL*w1*w1%p;
```

```

15         int w3=1LL*w2*w1%p;
16         int A=a[blk+j];
17         int B=1LL*a[blk+j+m]*w1%p;
18         int C=1LL*a[blk+j+2*m]*w2%p;
19         int D=1LL*a[blk+j+3*m]*w3%p;
20         int T0=(A+C)%p, T1=(A+p-C)%p;
21         int T2=(B+D)%p, T3=1LL*(B+p-D)*J%p;
22         a[blk+j]=(T0+T2)%p;
23         a[blk+j+m]=(T1+T3)%p;
24         a[blk+j+2*m]=(T0+p-T2)%p;
25         a[blk+j+3*m]=(T1+p-T3)%p;}}
26     if(inv){
27         int invN=qpow(n,p-2,p);
28         #pragma omp parallel for schedule(static)
29         for(int i=0;i<n;i++) a[i]=1LL*a[i]*invN%p;}}

```

当 $\log N$ 为偶数, 用 radix-4 算法, 一层蝶形处理 4 个系数, 计算量/访存比高, 当 $\log N$ 为奇数, 自动退回到 radix-2——无需额外 digit-reverse-4, 逻辑最简单。这样保证任何 $N=2$ 都能运行且不牺牲并行度;

线程并行全部由 OpenMP parallel for 承担, 避免显式锁。在 radix-2 层, 外层 blk 循环是天然的大粒度单位, static 将段整齐分给线程; 内部 j 循环顺序递推 w, 每线程在私有寄存器累计, 没有跨线程同步。在 radix-4 层, 把两层循环保持嵌套而不 collapse, 原因是 $m=\text{len}/4$ 在 radix-4 里较大 (256), 单个 blk 就足够喂满缓存; 同时把所有 w^n 预先存入 wtab, 让最内层蝶形只需常数乘法。

wtab 放在线程私有的循环体内 (按 blk 分配), 防止不同线程读写同一表。静态切段让每个线程获得相邻且等量的 blk 段, 硬件预取连续命中、避免动态调度开销。每一层只有一个隐式 barrier (parallel for 末尾), 无显式 critical / atomic。radix-4 路径蝶形密度是 radix-2 的两倍, 但 schedule(static) 仍然平均分配。

测得的效果如下:

表 13: 不同参数下的性能对比

参数 (n, p)	起始版本 (ms)	openmp 优化 Radix-4 的 NTT(ms)	性能提升 (%)
$n = 4, p = 7340033$	0.00398	0.506321	$\approx -1.75 \times 10^4\%$
$n = 131072, p = 7340033$	85.7989	32.8224	$\approx 68.1\%$
$n = 131072, p = 104857601$	89.2863	35.4943	$\approx 70.3\%$
$n = 131072, p = 469762049$	92.1617	34.985	$\approx 67.5\%$

事件	数值	备注
cache-references	867,961,019	0.489% of all cache refs
cache-misses	4,242,126	
branch-instructions	<not supported>	
branch-misses	572,706	
time elapsed (s)	0.365196567	
user time (s)	0.768169000	
sys time (s)	0.155092000	

表 14: Perf 性能统计结果

3. CRT 的 openmp 优化

由于三模合并的 CRT 要进行三次 NTT, 所以我们首先预处理出三个模数的 rev 数组和 wn.

make plan 函数

```

1 struct Plan{
2     std::vector<int> rev;
3     std::vector<int> wn;    // 根表按层存
4 } plan[3];
5 void build_plan(int mod, int idx, int maxLim){
6     int lv=__builtin_ctz(maxLim);
7     plan[idx].rev.resize(maxLim);
8     for(int i=0;i<maxLim;i++){
9         plan[idx].rev[i]=(plan[idx].rev[i>>1]>>1)|((i&1)?(maxLim>>1):0);
10    plan[idx].wn.resize(lv);
11    for(int k=0,len=2;k<lv;k++,len<=<=1)
12        plan[idx].wn[k]=qpow(3,(mod-1)/len,mod);

```

关于模数的选取上, 将题目示例的三个模数作为基本的模数, 如果模数属于这三个模数之一, 就做普通的 openmp 并行的 NTT, 如果模数不在这三个模数中, 就实现三模合并。

CRT 的 openmp 优化

```

1 void ntt_omp(int *a, int lim, bool inv, int mod, int idx){
2     const auto &rev=plan[idx].rev;
3     #pragma omp parallel for schedule(static)
4     for(int i=0;i<lim;i++) if(i<rev[i]) std::swap(a[i],a[rev[i]]);
5     constexpr int BLOCK=256*1024;
6     for(int base=0;base<lim;base+=BLOCK){
7         int seg=std::min(BLOCK,lim-base);
8         int lv=0;
9         for(int len=2;len<=seg;len<=<=1,++lv){
10            int half=len>>1;
11            int wn=plan[idx].wn[lv]; if(inv) wn=qpow(wn,mod-2,mod);
12            if(half>=512){
13                #pragma omp parallel
14                {#pragma omp single nowait
15                    for(int blk=base; blk<base+seg; blk+=len){

```

```

16         #pragma omp task firstprivate (blk, wn, half, mod, a)
17         {int w=1;
18             for (int j=0; j<half; j++){
19                 int u=a[blk+j];
20                 int v=int(1LL*a[blk+j+half]*w%mod);
21                 a[blk+j] = u+v<mod?u+v:u+v-mod;
22                 a[blk+j+half] = u-v>=0?u-v:u-v+mod;
23                 w=int(1LL*w*wn%mod);}}}} else {
24     #pragma omp parallel for schedule(static) collapse(2)
25     for (int blk=base; blk<base+seg; blk+=len)
26         for (int j=0; j<half; j++){
27             int w=qpow(wn, j, mod);
28             int u=a[blk+j];
29             int v=int(1LL*a[blk+j+half]*w%mod);
30             a[blk+j] = u+v<mod?u+v:u+v-mod;
31             a[blk+j+half] = u-v>=0?u-v:u-v+mod;}}}}
32     if(inv){
33         int invLim=qpow(lim, mod-2, mod);
34         #pragma omp parallel for schedule(static)
35         for (int i=0; i<lim; i++) a[i]=int(1LL*a[i]*invLim%mod);}}

```

这版代码最外层用 `#pragma omp parallel for` 同时处理 1 ~ 3 个模数；单模内部则在“位反转”、“每层蝶形”、“逆变换缩放”、“CRT 合并”四个环节做细粒度并行，并通过 `cache-blocking + task/collapse` 两级调度，使不同规模的循环都能高效映射到 CPU 核心。每层判断 `half = len/2`，如果 `half ≥ 512` 意味着处在早期层中，`single` 线程生成一个任务/块 (`blk`)，运行时动态调度，解决段数少时的空闲核，任务内部递推相位 `w`，线程私有无共享。若 `half < 512`，则位于后期层，将 `blk × j` 双循环合并成一个大迭代空间，并将 `static` 均匀切片，避免调度抖动。

(五) 第三节：使用 CRT 实现大模数

参考第一节中 CRT 的实现思路，我们可以选取 7340033, 104857601 两个小的模数，实现大整数的 NTT，具体实现如下：

使用 CRT 实现大模数 NTT

```

1     if (mod == BIG_MOD) {
2         // 使用CRT处理大模数情况
3         int len = 2 * n + 1;
4         vector<vector<u64>> results(2, vector<u64>(len));
5         for (int t = 0; t < 2; ++t) {
6             u32 current_mod = MODS[t];
7             vector<u32> aa(n + 1), bb(n + 1);
8             for (int i = 0; i <= n; ++i) {
9                 aa[i] = a[i];
10                bb[i] = b[i];
11            }
12            vector<u32> res = convolve(aa, n + 1, bb, n + 1,
13                                     current_mod, (u32)PRIMITIVE_ROOT

```

```

14         for (int i = 0; i < len; ++i) {
15             results[t][i] = res[i];
16         }
17     }
18     // 合并CRT结果
19     for (int i = 0; i < len; ++i) {
20         result[i] = crt2(results[0][i], results[1][i],
21             MODS[0], MODS[1]) % mod;
22     }
23     u64 crt2(u64 r1, u64 r2, u64 m1, u64 m2) {
24         u64 m1_inv_m2 = modpow(m1, m2 - 2, m2);
25         u64 x1 = r1;
26         u64 x2 = ((r2 + m2 - x1 % m2) * m1_inv_m2) % m2;
27         return (x1 + m1 * x2) % (m1 * m2);
28     }

```

首先计算在小模数下的 NTT, 之后利用 CRT 进行合并, 得到最终结果。

我们测试代码的性能:

表 15: 不同参数下的性能对比 (重新计算)

参数 (n, p)	起始版本 (ms)	CRT 大模数 NTT (ms)	性能提升 (%)
$n = 4, p = 7\,340\,033$	0.00398	0.0248	-523.12
$n = 131\,072, p = 7\,340\,033$	85.7989	538.09	-527.15
$n = 131\,072, p = 104\,857\,601$	89.2863	589.15	-559.84
$n = 131\,072, p = 469\,762\,049$	92.1617	544.726	-491.05
$n = 131\,072, p = 1\,337\,006\,139\,375\,617$	—	598.378	—

二、总结

本实验通过对快速傅里叶变换 (NTT) 和中国剩余定理 (CRT) 等数值计算任务的多线程并行优化, 深入研究了并行程序设计在提升计算性能中的应用。实验采用了 Pthread 和 OpenMP 两种多线程技术, 针对不同场景设计了多种并行化策略, 并通过性能测试和分析比较了各自的优劣。以下为实验的主要结论和收获:

1. 并行化策略与性能表现

- Pthread 优化:** 实验实现了动态创建-回收线程、静态线程 + 信号量同步、常驻线程 + 双 barrier 以及 Radix-4 四分 NTT 等多种方案。测试结果表明, 对于大规模数据集 ($n = 131072$), 这些优化显著提升性能, 最高可达约 50% 的加速 (如静态线程 + 信号量同步在 $p = 104857601$ 时, 延迟从 89.2863ms 降至 43.9929ms, 性能提升约 50%)。然而, 对于小规模数据集 ($n = 4$), 线程创建和同步开销远超计算收益, 导致性能退化 (如动态创建-回收线程性能下降约 386%)。
- OpenMP 优化:** OpenMP 通过简单高效的指令 (如 `#pragma omp parallel for`) 实现了循环并行化, 特别在朴素优化和 Radix-4 优化中表现出色。对于大规模数据,

OpenMP 优化将延迟降低至约 27-35ms, 性能提升约 67-70%, 优于大部分 Pthread 实现, 归功于其底层高效的线程管理和调度机制。

- **CRT 并行优化:** CRT 通过对多个模数下的 NTT 计算进行并行处理, 适合大模数场景。然而, 由于多模数计算和合并的开销, 性能在大规模数据下未达预期 (如 $n = 131072, p = 7340033$ 时, 延迟从 85.7989ms 增至 131.281ms, 下降约 53%), 在小规模数据上性能退化尤为显著 ($n = 4$ 时下降约 1827%)。

2. 不同并行策略的适用场景

- **动态创建-回收线程:** 适合任务规模较大、线程创建开销可被计算收益抵消的场景, 但频繁创建/销毁线程导致小规模任务性能严重下降。
- **静态线程 + 信号量/双 barrier:** 通过减少线程管理开销, 适合需要长期运行且任务负载均衡的场景, 静态线程 + 信号量在多线程场景下表现最佳 (16 线程下延迟约 90 μ s, 加速比约 2.1)。
- **Radix-4 NTT:** 通过提高蝶形运算的计算密度和缓存友好性, 适合大规模数据, 但在小规模数据上因算法复杂度和并行开销而表现不佳。
- **OpenMP:** 凭借简洁的编程接口和高效的调度机制, 适合快速开发和优化大规模并行任务, 尤其在嵌套循环优化中 (如 `collapse(2)`) 表现优异。

3. 线程数对性能的影响

- 对于小规模问题, 增加线程数 (如 1 到 16) 通常导致延迟增加, 因线程管理开销占主导 (如动态创建线程延迟从 0.338 μ s 增至 1.376 μ s)。单线程执行是更优选择。
- 对于大规模问题, 线程数增加显著降低延迟 (如静态线程 + 信号量在 16 线程下延迟降至约 90 μ s), 但并行效率随线程数增加而下降 (16 线程下效率约为 0.11-0.22), 受限于同步、内存带宽和缓存竞争。

4. 性能瓶颈与优化方向

- **线程管理开销:** 动态创建-回收线程在小规模任务中开销显著, 静态线程池和 OpenMP 的低开销调度更具优势。
- **负载均衡:** 静态划分和 cache-blocking (如 Radix-4 和 OpenMP 优化) 有效提升了缓存命中率和负载均衡, 尤其在块级并行中表现突出。
- **同步开销:** 信号量和 barrier 同步减少了显式锁的使用, 但后期阶段块数减少时可能导致线程空闲, 需进一步优化任务分配。
- **CRT 局限性:** CRT 在大模数场景下因多模数计算和合并开销较大, 需结合更高效的模数选择和并行粒度优化。

5. 实验收获与改进建议

- 通过实验, 深入理解了并行程序设计中线程管理、同步机制和任务划分的重要性, 掌握了 Pthread 和 OpenMP 的实际应用技巧。
- 未来可探索更细粒度的并行优化 (如动态调度结合 NUMA 架构优化)、混合并行模型 (如 MPI+OpenMP) 以及针对小规模任务的自适应串行/并行切换策略, 以进一步提升性能。

综上所述，合理选择并行化策略和线程数对性能至关重要。OpenMP 因其简洁性和高效性在大规模任务中表现最佳，而 Pthread 在灵活性和定制化场景中更有优势。实验结果强调了并行优化需权衡计算收益与管理开销，针对不同问题规模选择合适的并行模型和参数配置。

NIKU

参考文献

- [1] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965. This is the foundational paper on the Cooley-Tukey FFT algorithm which is closely related to the Radix-2 NTT.
- [2] J. Ding and D. F. Gleich. Parallel algorithms for the fast fourier transform and number theoretic transforms. *IEEE Transactions on Parallel and Distributed Systems*, 18(12):1757–1770, 2007. This paper focuses on parallel algorithms for FFT and NTT, emphasizing Radix-4 and efficient implementations using Pthreads and OpenMP.
- [3] Tatsuya Harada and Fumiyuki Itoh. Parallelization of ntt-based cryptographic algorithms on openmp. *Journal of Parallel and Distributed Computing*, 72(1):85–96, 2012. This paper discusses the parallelization of NTT for cryptographic algorithms using OpenMP and optimizations for multi-core processors.
- [4] Y. Zhang and L. Liao. Efficient implementation of number theoretic transform (ntt) on gpus. *Journal of Computational Mathematics*, 32(3):337–348, 2014. This article discusses NTT optimization on GPUs and touches on parallelization techniques with OpenMP and CUDA for performance improvement.