一、 问题定义

椭圆曲线密码学（Elliptic Curve Cryptography, 下文简称 ECC）是现代密码学中非对称加密中非常重要的一种基础理论。已经在安全领域中有了非常重要的应用，包括 ECC 公钥加密、数字签名、零知识证明等应用。但是，基于椭圆曲线结构的密码学应用在实际应用中的技术痛点也较为突出：虽然椭圆曲线公钥仅需要较低的位数就可以达到传统 RSA 公钥的安全性（256bit 的 ECDSA 签名安全性强于 2048bit 的 RSA 签名），节省了存储开销，这在许多对存储空间敏感的密码学应用（例如区块链技术）中是关键的；然而，计算和验证一个椭圆曲线签名（即计算椭圆曲线数乘）的时间开销却非常大，这影响了实际密码学应用的吞吐率。整个椭圆曲线的计算开销通常是取决于椭圆曲线上基于大素数域的乘法操作的时间，这也是椭圆曲线加密中的所依赖的难解性问题即计算：

$$Q = kP \qquad (1)$$

其中k一般为 256 位的大整数，P为椭圆曲线上的点的坐标，已知Q很难求解P。

二、 分析与实验

1.传统方法分析

整个椭圆曲线点乘计算过程主要是依赖于椭圆曲线的点加和倍点计算这两个基础操作完成。而一个大素数域上倍点计算目前 openSSL 的算法实现所消耗的时间约为 3M+5S（其中 M 为大整数乘法，S 为大整数平方），点加操作消耗约为 11M+5S，使用软件实现大整数乘法与加法，则有$O(M) \approx O(S)$，因此我们可以看到一个点加操作所消耗的时间约等于倍点消耗时间的两倍。采用传统 NAF 算法计算点乘：

输入: 正整数 k, $P \in E(F_q)$.

输出: kP

1. $Use\ NAF\ algorithm\ compute\ NAF(k) = \sum_{i=0}^{l-1} k_i 2^i$
2. $Q \leftarrow \infty$
3. $For\ i\ from\ l-1\ downto\ 0\ do$
   3.1 $Q \leftarrow 2Q$
   3.2 $If\ k_i = 1\ then\ Q \leftarrow Q + P$
   3.3 $If\ k_i = -1\ then\ Q \leftarrow Q - P$
4. $Return(Q)$.

从宏观上来看（即暂不细化点加与倍点的内部计算流程）我们可以容易看出其中需要l次倍点操作和约$\frac{l}{3}$次(取决于 NAF 的结构)的点加操作整个时间复杂

度为$O(8lM+\frac{16}{3}lM)$，空间复杂度为常数级别，只需要存储两个点的坐标即可。

2. 多线程并行优化

由于在 NAF 算法计算中点加操作依赖于相应的二倍点的结果，因此很难对该过程进行向量化，如果是采用循环累加的计算方式进行并行化，那么时间复杂度为$O(\frac{k}{p}16M)$，由于k约为 256 位的大整数，显然时间复杂度指数级，远远超过串行的算法。因此考虑采用传统多线程的方式进行任务划分，对倍点计算和点加计算进行划分，使用共享内存的通信方式共享中间的计算结果，很明显该并行方法的效果依赖于k的二进制表达中非零的数量，若非零的数量较多，由于倍点计算比点加计算要快很多，那么时间复杂度大概能够降为$O(8lm)$，空间复杂度变为$O(lP)$，即二倍点计算过程中需要存储每一次的结果以供点加计算任务使用。

```
vector<long> find_naf(bigint<n> &scalar, const size_t window_size) {
    const size_t length = scalar.max_bits(); // upper bound
    std::vector<long> res( n: length+1);
    bigint<n> c = scalar;
    long j = 0;
    while (!c.is_zero())
    {
        long u;
        if ((c.data[0] & 1) == 1)
        {
            u = c.data[0] % (1u << (window_size+1));
            if (u > (1 << window_size))
            {
                u = u - (1 << (window_size+1));
            }

            if (u > 0)
            {
                mpn_sub_1(c.data, c.data, n, u);
            }
            else
            {
                mpn_add_1(c.data, c.data, n, -u);
            }
        }
        else
```

图 1. 求解 NAF

```
EC_POINT *Q  = EC_POINT_new(group);
EC_POINT_set_affine_coordinates(group, Q, BN_new(), BN_new(), ctx);
print_ECP(group, Q, ctx);
auto res = find_naf( &: scalar,  window_size: 1);
// calc point multiplication
for (long i = res.size() - 1; i >= 0; i--) {
    EC_POINT_dbl(group, Q, Q, ctx);
    if (res[i] == 1) {
        EC_POINT_add(group, Q, Q, G, ctx);
    } else if (res[i] == -1) {
        EC_POINT_invert(group, G, ctx);
        EC_POINT_add(group, Q, Q, G, ctx);
        EC_POINT_invert(group, G, ctx);
    }
}
```

图 2. 串行 NAF 计算点乘

```
    thread process1(dbl_ECP, group, G, ctx);
    thread process2(add_ECP, group, G, ctx);
    process1.join();
    process2.join();
    return 0;
}

//
void dbl_ECP(EC_GROUP *group, EC_POINT*P, BN_CTX *ctx){
    EC_POINT * G = EC_POINT_new(group);
    EC_POINT_copy(G, P);
    for (int i = 0; i < naf_res.size(); ++i) {
        if (naf_res[i] == 1) {
            EC_POINT * newP = EC_POINT_new(group);
            EC_POINT_copy(newP, G);
            dbl_res[i] = newP;
        } else if (naf_res[i] == -1) {
            EC_POINT * newP = EC_POINT_new(group);
            EC_POINT_copy(newP, G);
            EC_POINT_invert(group, newP, ctx);
            dbl_res[i] = newP;
        }
        EC_POINT_dbl(group, G, G, ctx);
    }
    dbl_end = 1;
}
```

图 3、多线程方式计算点乘

三、实验结果分析与改进

1.实验结果正确性验证

输入：椭圆曲线坐标 P=(32C4AE2C1F1981195F9904466A39C9948FE30BBFF2660BE1715A4589334C74C7, BC3736A2F4F6779C59BDCEE36B692153D0A9877CC62A474002DF32E52139F0A0)

标量：k= 597875121111111187545464564512364987845555555555557897543645122318789794587

计算得到结果如表 1 所示，可以看出，DBC 实现计算结果与 NAF 算法一致。

表 1 正确性测试表

算法	结果
NAF	(50512AFE7CC71AFDE310E52ED3FF99C1B944DF8D9B59864A3E37D033CCA7EC5C, 79602067085C8B3FCADDB825E25E3263CE379F763778CD8CE06AB76F2E4DAFE0)
Multi-thread	(50512AFE7CC71AFDE310E52ED3FF99C1B944DF8D9B59864A3E37D033CCA7EC5C, 79602067085C8B3FCADDB825E25E3263CE379F763778CD8CE06AB76F2E4DAFE0)

本次作业实验环境为 Linux 系统发行版 ubuntu20.04，处理器为 AMD@Ryzen7 5800H，使用 C++语言实现，其中椭圆曲线库点加与倍点计算采用 openssl 库的实现，椭圆曲线参数与基点设置如表 2 所示，实验结果如表 3 所示。

表 2 椭圆曲线参数

椭圆曲线（sm2）	y2 = x3 + ax + b
素数 P	FFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFF
a	FFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFF00000000FFFFFFFFFFFFFFFC
b	28E9FA9E9D9F5E344D5A9E4BCF6509A7F39789F515AB8F92DDBCBD414D940E93
基点 G	(32C4AE2C1F1981195F9904466A39C9948FE30BBFF2660BE1715A4589334C74C7, BC3736A2F4F6779C59BDCEE36B692153D0A9877CC62A474002DF32E52139F0A0)

表 3 实验结果

n(bits)	r(M/S)	T_naf(ms)	T_multi(ms)	Speed up
150	1	0.418	0.335	1.3
200	1	0.434	0.337	1.3
250	1	0.447	0.348	1.3
300	1	0.465	0.361	1.3
350	1	0.476	0.344	1.4
400	1	0.481	0.343	1.4

实验数据均为 1000 次平均结果

分别对随机生成不同位的k进行测试，我们可以发现使用多线程对任务进行拆分并行计算点加与倍点计算，相比于传统的 NAF 算法有 1.3 倍左右的提升。本次实验主要是对采用双线程，还可以改进的地方还存在很多，例如对与倍点计算也可以使用多个线程分别从头和尾分别计算倍点，这样可以减少点加计算的等待时间。后续还可以深入分析倍点与点加计算内部的运算流程，对其中可以并行化的过程进行加速。