# ics-lab1 实验报告

181220010 丁豪

2019 年 12 月 19 日

## 任务1：实现 multimod

```
int64_t multimod_p1(int64_t a, int64_t b, int64_t m) {
  uint64_t ah = a & 0xffffffff00000000,
           al = a & 0x00000000ffffffff,
           bh = b & 0xffffffff00000000,
           bl = b & 0x00000000ffffffff,
           m2 = m;
  ah >>= 32;
  bh >>= 32;
  // [ahbh*2^64+(albh+ahbl)*2^32+albl ] mod m
  // u1 最多 (2^31-1)^2=2^62-2^32+1
  uint64_t u1=(ah*bh)%m2;
  for (int i=0;i<64;i++){
    u1 <<= 1;
    u1 %= m2;
  }

  // u2 最多 (2^31-1)^(2^32-1)=2^63-2^32-2^31+1
  uint64_t u2=(((ah*bl)%m)+((al*bh)%m))%m;
  for (int i=0;i<32;i++){
    u2 <<= 1;
    u2 %= m2;
  }
  uint64_t u3 = (al*bl)%m2; // 最后一项不会溢出

  return (int64_t)((((u1%m2)+(u2%m2))%m2)+(u3%m2))%m2;
}
```

基础公式,其中$h, l$下标表示的是高、低32位的意思

$$ab = (a_h b_h) * 2^{64} + (a_h b_l + a_l b_h) * 2^{32} + a_l b_l \tag{1}$$

$$a + b \ mod \ m = (a \ mod \ m + b \ mod \ m) \ mod \ m \tag{2}$$

$$a * 2^k \ mod \ m = [(a \ mod \ m) * 2] * 2^{k-1} \ mod \ m \tag{3}$$

算法总体思路是，既然直接计算64位乘64位会变成128位，那我们就把64位拆成高低32位分别乘，这样每一次乘法的结果都不会超过64位，完全在我们的数据类型允许范围之内。

具体实现中，分别计算(1)中三个部分的值，并使用(2)进行额外取模操作，就可以获得我们需要的$a * b \ mod \ m$的值。由于$a_h b_h, a_h b_l, a_l b_h$这三项产生了溢出，因而我们将结果写成2的科学计数法表示方式，并如(3)所示迭代执行乘二与取模运算，这样就保证了每一步都不会溢出。

作为对照，使用python程序来进行相同运算，并比较两者在不同输入下的结果，可以对程序正确性有一定信心。如下为正确实现样例。

0xffffff * 0xffffff % 0x7fffffffffff = 0x7fffffe00000002

0xffffffff * 0xffffffff % 0x7fffffffffffff = 0x7ffffe000000200

0xfffffffff * 0xfffffffff % 0x7fffffffffffff = 0x7ffffe0000020000

0xfffffffff * 0xfffffffffffff % 0x7fffffffffffff = 0x6fffff2000000000

0xfffffffffffff * 0xfffffffffffff % 0x7fffffffffffff = 0x6200000000000000

0x7fffffffffffff * 0x7fffffffffffff % 0x5fffffffffffff = 0x4aaaaaaaaaaaaaaa

## 任务2：性能优化

```
int64_t multimod_p2(int64_t a, int64_t b, int64_t m) {

  uint64_t aa=a,bb=b,mm = m;
  uint64_t unit = aa % mm;
  uint64_t answer = 0;

  // 把bb逆序为cc
  uint64_t cc=0;
  while(bb){
    cc <<= 1;
    cc += (bb&1);
    bb >>= 1;
  }
  // 将原bb最高位放到现在的最低位
  while(!cc&1)
    cc >>= 1;

  // 展开计算
  while (cc>1){
    if (cc&1){
      answer += unit;
      answer %= mm;
    }
    answer <<= 1;
    answer %= mm;
    cc >>= 1;
  }
  // 对第一位
  if (cc&1){
      answer += unit;
      answer %= mm;
  }

  return (int64_t)answer;
}
```

新的算法使用了如下公式

$$ab = (\Sigma_{i=0}^{62} a_i * 2^i)b \tag{4}$$

其实在P1算法中，我们就已经使用了类似的思想，那便是将乘法与取模交替执行以此来将中间结果控制在64位之内。在这里我们根据提示了解到需要把a直接分解到每一个比特，如此一来乘法操作和2的指数上升，实际上一经可以变成简单的加法与位运算。

我们自顶向下，逐次求2的最高次幂所对应项与m的模，然后把结果记下来，在下一轮迭代的时候加到2的$n-1$次幂对应的项上，如此重复直到$a_1$，在$a_0$处终止，此时不必再乘2（左移）。至于如何实现从a的高位到低位逐位给b进行操作，我们先把一个操作数倒序，然后就可以从低位到高位操作，这样又能用位运算来简单解决了。（由于公式与算法中a和b的含义是反的，所以我们就姑且称之为两操作数）

有关时间复杂度的分析，我们直接在一个测试函数中，分别调用p1 p2两个算法计算相同a,b,m相同次数(较多)，并比较两者耗费的时钟周期数量。由于计算数量较多，根据大数定律，可以认为偶然性误差较低，数值与期望相近。为了避免编译器优化，使用volatile关键字。


通过观察每次个体的差异不超过自身数量的百分之十，因而我们便没有统计均值，仅进行了数轮测试。

```
constantine@debian:~/ics-workbench/multimod$ gcc -O0 Mytest.c p1.c p2.c p3.c
constantine@debian:~/ics-workbench/multimod$ ./a.out
p1 answer:0x19249249249249 time:1194128
p2 answer:0x19249249249249 time:1678167
constantine@debian:~/ics-workbench/multimod$ gcc -O1 Mytest.c p1.c p2.c p3.c
constantine@debian:~/ics-workbench/multimod$ ./a.out
p1 answer:0x19249249249249 time:1018739
p2 answer:0x19249249249249 time:1297389
constantine@debian:~/ics-workbench/multimod$ gcc -O2 Mytest.c p1.c p2.c p3.c
constantine@debian:~/ics-workbench/multimod$ ./a.out
p1 answer:0x19249249249249 time:1015576
p2 answer:0x19249249249249 time:1395703
```

出乎意料的是，p2中使用的新算法在时间测试中尚且不如P1中使用的算法，两者相差不多，约莫在渐进意义上只差常数。出现这样的结果其实并不非常意外，简单分析p1与p2两个算法我们不难发现，他们的渐进时间复杂度都是O(nlgn)，但是如果把乘法看做O(1)时间，那么p1算法的实际时间开销是O((64+32)*64)，而p2是O((64*4)*64)，这也就说得通了。

## 任务3：解析神秘代码

```
int64_t multimod_fast(int64_t a, int64_t b, int64_t m) {
  int64_t t = (a * b - (int64_t)((double)a * b / m) * m) % m;
  return t < 0 ? t + m : t;
}
```

IEEE754标准中64位浮点数的尾数只有52位，并设置隐藏位1在小数点之前，所以实际上可以表示53位有效数字。同时参考C语言标准中，将整形强制类型转换成浮点型的原则——逐渐将最低有效为置0，所以(double)a实现的效果等同于取a的至多高53位有效数字，并将后续位设置为0。同理在进行乘法整除时另一个操作数也会被按照这种规则转换成double型，于是我们始终是对每一步中间结果取高53位的效果。于是乎(double)a*b得到的是a*b真实值的至多 53位有效数字解，他的大小在$2^{126}$之内。至于后面的/m我们稍后再说。

浮点型强制类型转换成整形的方式，是直接舍去小数部分，保留整数部分。由于这里全部是整数运算，并没有产生小数，同时double为53位有效数字，而Int64_t有63位有效数字，因此这次(int64_t)转换是无损的。

我们简单观察式子所表达的数学含义，可以发现其实最外层括号内的值在纯数学意义上是取a*b/m的余数，也就是 a*b mod m。因此如果我们

能保证(int64_t)((double)a*b/m)能正确得到与$ab/m$的结果（这里的除为整除），那么将做差两者的结果全部保留低64位时可以得到余数的正确值（由于高位为1的关系，或许会+或-$2^64$,不过这一影响将在截断为64位的过程中被去除），那么最终结果的正确性就可以保证。

所以a,b,m满足的条件，就是使得(double)a*b/m结果与ab/m数学结果相同的条件, 其中显而易见的是a*b不会溢出的情况，当ab会溢出时，经过推导我认为是

$$ab/h \leq 2^{54} - 1 \tag{5}$$

如果它满足，则整除结果完整保留，如果它不满足，则整除结果失真，会导致最终结果误差。

测试过程使用python脚本随机生成样本数据a,b,m,answer四元组，并将他们按行存储在文本文件中。在测试程序中读取此文件，并把C语言算法获得的结果与python直接生成的answer值进行对比，进而可以知悉C程序计算是否正确。

```
constantine@debian:~/ics-workbench/multimod$ gcc -O0 Mytest.c p1.c p2.c p3.c
constantine@debian:~/ics-workbench/multimod$ ./a.out
p1 answer:0x19249249249249 time:1168566
p3 answer:0x19249249249249 time:21026
constantine@debian:~/ics-workbench/multimod$ gcc -O1 Mytest.c p1.c p2.c p3.c
constantine@debian:~/ics-workbench/multimod$ ./a.out
p1 answer:0x19249249249249 time:975899
p3 answer:0x19249249249249 time:16467
constantine@debian:~/ics-workbench/multimod$ gcc -O2 Mytest.c p1.c p2.c p3.c
constantine@debian:~/ics-workbench/multimod$ ./a.out
p1 answer:0x19249249249249 time:962751
p3 answer:0x19249249249249 time:16844
```

性能比较方法同上,不过把multimod_p2换成了multimod_p3,由于p1与p2的比较在上面已经做了，这里只进行p1与p3的性能比较.可以发现二者的时间开销大约在50倍左右，相当于O(n)与O(1)的对比（n取64时），十分合情合理。

## 4：结束语

本次实验给我的感觉，计算机系统知识较少，主要只有IEEE754浮点数表示方法，整形表示方法，两者互相强制类型转换的细节。而代数方面反倒涉及较多，这是我较为不擅长的，因此在上文中难免有纰漏，如若不吝赐教将不胜感激。

实验中自己编写的测试程序已于提交阶段全部删除，希望不会给自动化批阅系统带来不必要的麻烦。