Notes on the complexity of CRT

Niels Möller

January 18, 2007

Abstract

These notes describe preliminary results in the search for the optimal way to reconstruct a number from its remainders modulo some set of smaller numbers, using the Chinese Remainder Theorem. The case of interest is when we have a small number of small moduli, known in advance. The complexity is measured as the number of multiplications of single words. Of the methods considered, it turns out that for up to 5 moduli, Garner's method is the best. The case of 6 moduli is special, in that one can save one multiplication by splitting the modulo in two groups of for and 2 each. From seven moduli and up, the best method is to compute the value as a linear combination of base numbers that are of the same size as the result. All these methods are asymptotically $O(n^2)$, where n is the number of moduli, but the methods described in here use approximately $3.86(n-1)^1.57$ multiplications for $1 \leq n \leq 10$. When the number of moduli is very large, it is expected that a divide and conquer strategy, taking advantage of subquadratic multiplication such as Karatsuba or Schönhage-Strassen; this is not analyzed in detail in the present notes, since the focus is on fairly small numbers.

1 Introduction

The Chinese remainder theorem says that given n pairwise coprime moduli m_i, and n numbers x_i, there exists a number x such that $x = x_i \pmod{m_i}$ for all i. The number x is also uniquely determined $\pmod{M}$, where $M = m_1 m_2 \cdots m_n$. These notes discuss the complexity of actually computing this x.

Each x_i can be thought of as an element of Z_{m_i}, and x as an element of Z_M, and this is a nice algebraic relation which I don't discuss further.

Instead, we assume that each m_i fits in a single computer word, then x_i is represented as a computer word with a value in the range $0 \leq x_i < m_i$. The solution x can be represented in n computer words.

We are looking for an algorithm that takes x_i as input and outputs x. The m_i are treated as fixed constants: Precomputations involving only the m_i are free.

The primary interest is in computations with fairly small n. The computational cost is based on the number of basic multiplications needed, where a *basic multiplication* takes two computer words as inputs, and produces two computer words representing their (unsigned) product. The cost of additions, conditionals, etc is neglected.

1.1 Division using multiplication

The algorithms will need reductions modulo m, and for efficiency, this should be computed using basic multiplications. Montgomery multiplication is widely used, but somewhat impractical in our application. Instead, we will use preinverses (Granlund and Montgomery [3]) which is simpler and almost as good.

For single word operands, $ab \bmod m$ can be computed in three basic multiplications using a precomputed single-word inverse for m: the first for computing ab, the second to compute the approximate quotient, and the third and final to compute the remainder. See the appendix for more details.

We also need to perform this computation when the inputs are represented as multiple (but few) words. Assume that a is k words, b is ℓ words, and m is n words, with $k+\ell \geq n$, and we use school-book multiplication. Then ab takes $k\ell$ basic multiplications. For the division, the quotient is $k+\ell-n$ words and the remainder is n words. An approximation q of each quotient word is computed in one basic multiplication, using an inverse of the most significant word of m, and then to subtract qm we need n more basic multiplications. So the division takes $(k+\ell-n)(n+1)$ basic multiplications, and the total for multiplication ab and division is thus

$$k\ell + (k+\ell-n)(n+1)$$

In case $k=\ell=n$, this reduces to $2n^2+n$.

2 Some quadratic algorithms

2.1 Linear combination

The simplest way to compute x is to express it as a linear combination in Z_M. Define $M'_j = \prod_{i\neq j} m_i$ and

$$c_j = M'_j \left({M'_j}^{-1} \pmod{m_j} \right) \tag{1}$$

Then the c_j have the property that $c_j = \delta_{ij} \pmod{m_i}$. It follows that

$$x = \sum c_i x_i \pmod{M}$$

This can be computed by first computing and summing all terms (n^2 basic multiplications), and then reducing the sum by M. The sum is at most $\sum c_i(m_i - 1)$. For some choices of m_i, this maximum value may fit in $n+1$ words; in general it is bounded by $M \sum m_i$ which fits in $n+1$ words and $\log_2 n$ bits.

For the fairly general case that the maximum value fits in $n+2$ words (always true if the number of moduli, n, is a single word), the quotient is two words, which implies that the final reduction can be done using $2(n+1)$ basic multiplications, for a total of

$$n^2 + 2n + 2 \tag{2}$$

multiplications.

Also keep in mind the special case that $\sum c_j(m_j-1)$ fits in $n+1$ words. Then x can be computed in only

$$n^2 + n + 1 \tag{3}$$

multiplications. This is actually pretty good, so let's call this the "good luck" algorithm, and include this in the algorithm comparison at the end.

2.2 Cohen's algorithm

Cohen (A course in computational algebraic number theory [1], Algorithm 1.3.11) gives the following algorithm.

1. For $j = 2, \dots, n$, set
$$c_j = (m_1 m_2 \cdots m_{j-1})^{-1} \pmod{m_j}$$
2. For $j = 2, \dots, n$, set
$$y_j = (x_j - (x_1 + m_1(y_2 + m_2(y_3 + \cdots + m_{j-2}y_{j-1})\cdots)))c_j \bmod m_j$$
3. Output
$$x = x_1 + m_1(y_2 + m_2(y_3 + ...m_{n-1}y_n)\cdots)$$

Step 1 are precomputations.

Step 2 uses modular multiplications on single-word values. To compute $ab \bmod m$ takes three basic multiplications. y_j needs $j-1$ such operations, so in total for this step we need $3n(n-1)/2$ multiplications.

Step 3 multiplies increasingly large numbers by single-word numbers. The total number of basic multiplications in this step is $n(n-1)/2$. The grand total is thus $2n(n-1) = 2n^2 - 2n$ multiplications.

2.3 Iterative algorithm

Another variant is to add one moduli at a time. Let M_j be the sequence of partial products, $M_j = m_1 m_2 \cdots m_j$, and compute the sequence of X_j, $0 \le X_j < M_j$ such that $X_j = x_i \pmod{m_i}$ for $i = 1, 2, \dots, j$. We naturally have $X_1 = x_1$. For $2 \le j < n$, precompute
$$C_j = M_{j-1}^{-1} \pmod{m_j} \tag{4}$$
and compute
$$\begin{aligned} y_j &= (x_j - X_{j-1})C_j \bmod m_j \\ X_j &= X_{j-1} + M_{j-1}y_j \end{aligned}$$

This algorithm is described in Crandall and Pomerance, Prime numbers—a computational perspective [2], Algorithm 2.1.7, where it is attributed to Garner. It needs
$$\frac{1}{2}(n-1)(3n+2) \tag{5}$$
basic multiplications.

2.4 Summary

We see that Garner's algorithm is more efficient than Cohen's, for all n (except $n = 2$ where they are equivalent). For $n \ge 7$, the linear combination method is more efficient than Garner's method (or if we have "good luck" with our parameters, this method is more efficient already for $n \ge 5$.

n	Cohen	Garner	Linear	Good luck
2	4	4	10	7
3	12	**11**	17	13
4	24	**21**	26	21
5	40	**34**	37	*31*
6	60	50	50	*43*
7	84	69	**65**	*57*
8	112	91	**82**	*73*
9	144	116	**101**	*91*
10	180	144	**122**	*111*

Table 1: Cost for the basic quadratic algorithms. Bold figures are marks the best method among Cohen's Garner's and the linear combination method. The slanted figures in the rightmost column marks the lines where the linear combination is the most efficient, assumingt that the parameters satisfy the good luck condition.

3 Divide-and-conquer

Another alternative is to use a divide-and-conquer scheme. First, consider the example $n = 4$. Put $M_1 = m_1m_2$, $M_2 = m_2m_3$, and $C_2 = M_1^{-1} \pmod{M_2}$.

Then x can be found by first computing $X_i = x \mod M_i$ for $i = 1, 2$, by applying the method for $n = 2$ to the pairs (m_1, m_2) and (m_3, m_4). Next, use the same method to (M_1, M_2), by setting $Y_2 = (X_2 - X_1)C_2 \bmod M_2$ and then $x = X_1 + M_1Y_2$.

The values X_i, M_i, C_2, and Y_2 are all two-word numbers. To compute the product $(X_2 - X_1)C_2$ takes four basic multiplications, and the result is a four-word number which has to be reduced $\pmod{M_2}$ to get Y_2. This reduction can be computed using six basic multiplications. This method can be summarized as follows.

$C_2 = M_1^{-1} \pmod{M_2}$	precomputation
$X_1 = \mathrm{crt}_2(x_1, x_2, m_1, m_2)$	4 multiplications
$X_2 = \mathrm{crt}_2(x_3, x_4, m_3, m_4)$	4 multiplications
$Y_2 = (X_2 - X_1)C \bmod M_2$	10 multiplications
$x = X_1 + M_1Y_2$	4 multiplications

The grand total is thus 22 multiplications, two less than with Cohen's method, but one more than Garner's method.

This subdivision can be generalized in two ways. For very large $n = 2^k$, and assuming subquadratic multiplication, divide-and-conquer results in the running time $\leq 2kM(n)$, where $M(n)$ is the time needed for multiplication of two n-word numbers.

In the following, we continue to focus on fairly small n, and we assume that multiplication uses the schoolbook-style method, multiplying two n-word numbers using n^2 basic operations. Let $T(n)$ denote the operation count for crt_n,

Generalizing the method for $n = 2 + 2$ above, assume that $n = n_1 + n_2$. It turns out that since the computation of $Y_2 = (X_2 - X_1)C_2 \bmod M_2$ is more complex than the output computation $x = X_1 + M_1 Y_2$, it's desirable to have $n_1 \geq n_2$.

For the modular multiplication $(X_2 - X_1)C_2 \mod M_2$, we first reduce X_1 mod M_2, which takes $(n_1 - n_2)(n_2 + 1)$ operations. Then the inputs and outputs are of size n_2, so the rest of the operation takes $2n_2^2 + n_2$ operations. The final multiplication $M_1 Y_2$ takes $n_1 n_2$ multiplications. So we get

$C_2 = M_1^{-1} \pmod{M_2}$	precomputation
$X_1 = \mathrm{crt}_{n_1}(x_1, \dots x_{n_1}, m_1, \dots m_{n_1})$	$T(n_1)$ multiplications
$X_2 = \mathrm{crt}_{n_2}(x_{n_1+1}, \dots, x_n, m_{n_1+1}, m_n)$	$T(n_2)$ multiplications
$Y_2 = (X_2 - X_1)C \bmod M_2$	$n_1 + n_1 n_2 + n_2^2$ multiplications
$x = X_1 + M_1 Y_2$	$n_1 n_2$ multiplications

In total,

$$T(n_1) + T(n_2) + n_1 + n_2(2n_1 + n_2)$$

multiplications. In case $n_1 = n_2 = n/2$, the expression reduces to $2T(n/2) + n/2 + 3n^2/4$

Generalize this method a little bit further, assume that $n = n_1 + n_2 + n_3$, $M_1 = m_1 \cdots m_{n_1}$, $M_2 = m_{n_1+1} \cdots m_{n_1+n_2}$, $M_3 = m_{n_1+n_2+1} \cdots m_n$ and $X_j = x \pmod{M_j}$. We get

$C_2 = M_1^{-1} \pmod{M_2}$	precomputation
$C_3 = (M_1 M_2)^{-1} \pmod{M_3}$	precomputation
$X_1 = \mathrm{crt}_{n_1}(x_1, \dots x_{n_1}, m_1, \dots m_{n_1})$	$T(n_1)$
$X_2 = \mathrm{crt}_{n_2}(x_{n_1+1}, \dots, x_{n_1+n_2}, m_{n_1+1}, m_{n_1+n_2})$	$T(n_2)$
$X_3 = \mathrm{crt}_{n_3}(x_{n_1+n_2+1}, \dots, x_n, m_{n_1+n_2+1}, m_n)$	$T(n_3)$
$Y_2 = (X_2 - X_1)C_2 \pmod{M_2}$	$n_1 + n_1 n_2 + n_2^2$
$X_2' = X_1 + Y_2 M_1$	$n_1 n_2$
$Y_3 = (X_3 - X_2')C_3 \pmod{M_3}$	$n_1 + n_2 + (n_1 + n_2)n_3 + n_3^2$
$x = X_2' + y_3 M_2$	$(n_1 + n_2)n_3$

So the grand total is

$$T(n_1) + T(n_2) + T(n_3) + 2n_1 + n_2 + 2(n_1 n_2 + n_1 n_2 + n_2 n_3) + n_2^2 + n_3^2$$

In case $n_1 = n_2 = n_3 = n/3$, this reduces to $3T(n/3) + n + (8/9)n^2$.

Can this divide-and-conquer method be more efficient the methods of the previous section? To answer that, one can perform an exhaustive search over the ways to divide numbers of interest. It turns out, that $n = 6$ is the only (small) number where divide and conquer is beneficial. We have the following subdivisions:

- $6 = 5 + 1$ gives $T(5) + 16 = 50$ (equivalent to Garner's method).
- $6 = 4 + 2$ gives $T(4) + T(2) + 24 = 49$.

CPU	`crt2`	`crt3`	`crt4`	`crt5`
AMD Duron (32 bit)	31	119	236	387
AMD Opteron (64 bit)	32	98	195	313

Table 2: Actual running time, in cycles.

n	Operations	Strategy	Good luck
2	4	Garner	7
3	11	Garner	13
4	21	Garner	21
5	34	Garner	31
6	49	$4+2$	43
7	65	Linear	57
8	82	Linear	73
9	101	Linear	91
10	122	Linear	111

Table 3: Optimal subdivisions of n, and the resulting number of multiplications to compute the crt_n. The values above were found by exhaustive search over subdivisions of n into two or three pieces, and comparing to the direct quadratic methods. The rightmost column gives the operation count for the "good luck" algorithm, which is applicable only to some choices of m_i.

- $6 = 3 + 3$ gives $2T(3) + 30 = 52$,
- $6 = 4 + 1 + 1$ gives $T(4) + 29 = 50$
- $6 = 3 + 2 + 1$ gives $T(3) + T(2) + 35 = 50$.
- $6 = 2 + 2 + 2$ gives $3T(2) + 38 = 50$.

Hence, with the division $6 = 4 + 2$ we get $T(6) = 49$, one multiplication less than Garner's method.

In case we have good luck with the parameter choices, things are slightly different. For example, if $T(5) = 31$, then the subdivisions $6 = 5 + 1$ and $7 = 5 + 2$ are the best, and they result in $T(6) = 47$ and $T(7) = 64$.

4 Implementation

Table 2 gives the running rime, in cycles, for the reconstruction.

5 Summary and conclusions

Table 5 gives the optimal subdivision for some more values of n.

5.1 Conclusions

- For $n \leq 5$, Garner's method is optimal.
- $n = 6 = 4 + 2$ is the only with an interesting optimal subdivision.

- For $n \geq 7$, the linear combination in Z_m ($n^2 + 2(n+1)$ multiplications) appears to be optimal.
- Traditional divide-and-conquer, splitting the input into roughly equal parts, is *not* optimal. (If n is large enough so that subquadratic multiplication is available, then splitting into roughly equal parts can be expected to be more efficient).
- For $n \geq 5$, the "good luck" algorithm is better than the "optimal" one. Hence, one should check if one's moduli satisfy the "good luck" condition, or of some subset of size five or more moduli does. If so, this should be taken into account, and due to the divide-and-conquer strategy, such good luck may affect the optimal strategy also for larger n. For example, if $n = 5$ is a lucky number, then $T(5)$, $T(6)$, and $T(7)$ are reduced by 3, 2, and 1 multiplications, respectively.
- Since $T(1) = 0$, it is natural to measure complexity in terms of $n-1$. When doing so, the operation count for small n is appearantly subquadratic; e.g., $T(n) \approx 3.86(n-1)^{1.57}$ is quite accurate for $n \leq 10$. This is remarkable, since we assume quadratic schoolbook-style multiplication, and there's no Karatsuba-like procedure in the algorithm.
- However, for all variants considered here, $T(n) \geq n^2$, which indicates that subquadratic multiplication is necessary to achieve subquadratic performance for large n.

References

[1] Henri Cohen. *A course in computational algebraic number theory.* Springer, 1996.

[2] Richard Crandall and Carl Pomerance. *Prime numbers — a computational perspective.* Springer, 2001.

[3] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the SIGPLAN PLDI'94 Conference*, June 1994.

Appendix: Division by constants

It is well known that division by a constant can be performed using two multiplications and a few branches [3]. Assume that the word size is ℓ bits, and put $\omega = 2^\ell$. Consider the quotient $q = \lfloor n/d \rfloor$, where d is constant in the range $\omega/2 \leq d < \omega$, and $n = n_0 + n_1\omega$, with $0 \leq n_0 < \omega$, $0 \leq n_1 < d$. Then q fits in a single word.

First precompute an approximate inverse, $d' = \lfloor (w^2-1)/d \rfloor - \omega$ (the quotient $\lfloor (w^2-1)/d \rfloor$ is precisely $\ell + 1$ bits, so this definition implies that d' fits in a single word.

The simplest way of using the inverse d' is to compute an approximative quotient and remainder:

$$q' = \lfloor n_1(\omega + d')/\omega \rfloor = \lfloor n_1 d'/\omega \rfloor + n_1 \tag{6}$$

$$r' = n_0 + n_1\omega - dq' \tag{7}$$

Then it can be shown that $0 \leq r' < n_0 + 2d$, which implies that $r' < 4d$ and that $r' < 3\omega$. So the true quotient and remainder are found after at most three additional subtractions.

Of the two multiplications, the computation of q' uses only the most significant half of the product, but in practice, this is no cheaper to compute than the full 2ℓ bit product. For the second multiplication, most (but not all!) of the most significant half cancels in the subtraction. Hence it is sufficient to compute the least significant $\ell + 2$ bits of the product, but again, this is in practice no cheaper than computing the full product.

However, it's possible to do the computation using one full product, and one product that uses only the least significant ℓ bits. Such a small product is usually cheaper to compute, both on the x86 architecture (`mull` vs `imul`) and on RISC architectures. This method works as follows.

$$p = n_1 d' + n_0 + n_1\omega \tag{8}$$

$$q_0 = p \bmod \omega \tag{9}$$

$$q_1 = 1 + \lfloor p/\omega \rfloor \tag{10}$$

$$r' = (n_0 - q_1 d) \bmod \omega \tag{11}$$

Then $q_0 - \omega < n - q_1 d < q_0$, which is an interval smaller than ω; hence, the value is uniquely determined by its value modulo ω, i.e., r'. To find the true quotient, we must examine several different cases.

- If $d \leq r' < q_0$, then $q = q_1 + 1$.
- Else, if $q_0 \leq r' < (\omega - d)$, then $q = q_1 - 2$.
- Else, if $r' \geq q_0$, then $q = q_1 - 1$.
- Otherwise, $q = q_1$.

When d is close to ω, the first two cases are unlikely.