Faster computation of roots of polynomials over $\mathbb{F}_q$

Michael Monagan
Simon Fraser University

ICIAM minisymposium on
Polynomial and Computational Challenges in Computer Algebra

Vancouver, July 18th, 2011

A polynomial interpolation problem

Application [MJ 2010]: to interpolate a polynomial in 12 variables of degree 30 with t non-zero terms modulo a 32 bit prime p we need to compute the roots of $\Lambda(z) \in \mathbb{F}_p[z]$ of degree t using [Rabin 1980] where $\Lambda(z)$ has t roots in $\mathbb{F}_p$.

	1 core				4 cores	
t	time	roots	solve	probes	time	speedup
1019	7.94	0.65	0.08	5.76	2.58	(3.08x)
2041	31.3	2.47	0.32	22.7	9.94	(3.15x)
4074	122.3	9.24	1.26	90.0	38.9	(3.14x)
8139	484.6	34.7	5.02	357.3	152.5	(3.18x)

Cilk timings in CPU seconds on an Intel Corei7

Ahmdal's law ($t = 8139$): speedup ≤ 3.21 (4 cores) and ≤ 6.31 (12 cores).

We parallelized the solve time and reduced the roots sequential time from 34.7s to 10.4s (classical) to 2.25s (FFT) then 1.5s (GCD):

Ahmdal's law ($t = 8139$): speedup ≤ 3.96 (4 cores) and ≤ 11.58 (12 cores).

A polynomial interpolation problem

Application [MJ 2010]: to interpolate a polynomial in 12 variables of degree 30 with t non-zero terms modulo a 32 bit prime p we need to compute the roots of $\Lambda(z) \in \mathbb{F}_p[z]$ of degree t using [Rabin 1980] where $\Lambda(z)$ has t roots in $\mathbb{F}_p$.

	1 core				4 cores	
t	time	roots	solve	probes	time	speedup
1019	7.94	0.65	0.08	5.76	2.58	(3.08x)
2041	31.3	2.47	0.32	22.7	9.94	(3.15x)
4074	122.3	9.24	1.26	90.0	38.9	(3.14x)
8139	484.6	34.7	5.02	357.3	152.5	(3.18x)

Cilk timings in CPU seconds on an Intel Corei7

Ahmdal's law ($t = 8139$): speedup ≤ 3.21 (4 cores) and ≤ 6.31 (12 cores).

We parallelized the solve time and reduced the roots sequential time from $34.7s$ to 10.4s (classical) to 2.25s (FFT) then 1.5s (GCD):

Ahmdal's law ($t = 8139$): speedup ≤ 3.96 (4 cores) and ≤ 11.58 (12 cores).

Fast division in $F[x]$

Let $a, b \in F[x]$.
Compute the quotient q and remainder r of $a \div b$ such that

$$a = bq + r.$$

Let $b \;= b_0 + b_1 x + \cdots + b_d x^d$ so $d = \deg b$.
Let $b_r = b_d + \cdots + b_1 x + b_0 x^d$ be the reciprocal polynomial.

1: compute b_r^{-1} to $O(x^{d_q+1})$ with a Newton iteration ...$2M(d)$
2: compute $q_r = \lfloor a_r b_r^{-1} \rfloor_{d_q}$ then .... $1M(d)$
3: compute $r = a - bq$ .... $1M(d)$

Inverse using a Newton Iteration

Input: $d \in \mathbb{N}$ and $b = b_0 + b_1 x + \cdots \in F[x]$.
Compute $y = b^{-1}$ to $O(x^d)$

1 **if** $d = 1$ **return** b_0^{-1}.
2 compute $y = b^{-1}$ to $O(x^{\lceil d/2 \rceil})$ recursively.
3 **return** $(2y - y^2 b) \mod x^d$.

MCA: $T(d) = T(\frac{d}{2}) + M(\frac{d}{2}) + M(d) + O(d) \implies T(d) < \mathbf{3} M(d)$

FFT: $T(d) = T(\frac{d}{2}) + \underbrace{3FFT(2d)}_{\equiv \mathbf{1} M(d)} + O(d) \implies T(d) < \mathbf{2} M(d)$

Inverse using a Middle Product

3 **return** $2y - yb^2 \mod x^d$.

3 **return** $y + y(1 - yb) \mod x^d$.

$$yb = 1 + 0x + \cdots + 0x^{\frac{d}{2}-1} + \underbrace{\square x^{\frac{d}{2}} + \cdots + \square x^{d-1}}_{\text{middle product}} + \underbrace{\square x^d + \cdots + \square x^{\frac{3}{2}d-2}}_{\text{junk}}$$

HQZ [2002]:

$$T(d) = T(\tfrac{d}{2}) + M(\tfrac{d}{2}) + \overbrace{\mathbf{MP}(\tfrac{\mathbf{d}}{\mathbf{2}})}^{\equiv \mathbf{1} M(\frac{d}{2}))} + O(d) \implies T(d) < \mathbf{2} M(d)$$

FFT:

$$T(d) = T(\tfrac{d}{2}) + \overbrace{3FFT(\tfrac{3}{2}d)}^{\equiv M(\frac{3}{2}d)} + O(d) \implies T(d) < \tfrac{\mathbf{3}}{\mathbf{2}} M(d)$$

Rabin's 1980 root finding algorithm over $\mathbb{F}_q$

Input: p an odd prime, $f = 1\,x^d + \cdots + f_1 x + f_0 \in \mathbb{F}_p[x]$, $f_0 \neq 0$
Output: the roots of $f(x)$ in $\mathbb{F}_p$.

Lemma (Fermat)

Over $\mathbb{F}_p$, $x^{p-1} - 1 = \prod_{i=1}^{p-1} (x - i) = (x^{(p-1)/2} - 1)(x^{(p-1)/2} + 1)$

1. Compute $b = \gcd(x^{p-1} - 1, f)$ = all linear factors of f.
2. If $\deg b > 1$ compute $h = \gcd((x + \alpha)^{(p-1)/2} - 1, b)$
   for random $\alpha \in \mathbb{F}_p$ until h splits b.
   Then compute the roots of h and b/h recursively.

How do we compute $h = \gcd(a^m + c, b)$?
First compute $a^m \mod b$ using square-and-multiply.

Rabin's 1980 root finding algorithm over $\mathbb{F}_q$

Input: p an odd prime, $f = 1\,x^d + \cdots + f_1 x + f_0 \in \mathbb{F}_p[x]$, $f_0 \neq 0$
Output: the roots of $f(x)$ in $\mathbb{F}_p$.

Lemma (Fermat)

Over $\mathbb{F}_p$, $x^{p-1} - 1 = \prod_{i=1}^{p-1}(x - i) = (x^{(p-1)/2} - 1)(x^{(p-1)/2} + 1)$

1. Compute $b = \gcd(x^{p-1} - 1, f)$ = all linear factors of f.
2. If $\deg b > 1$ compute $h = \gcd((x + \alpha)^{(p-1)/2} - 1, b)$ for random $\alpha \in \mathbb{F}_p$ until h splits b.
   Then compute the roots of h and b/h recursively.

How do we compute $h = \gcd(a^m + c, b)$?
First compute $a^m \mod b$ using square-and-multiply.

Algorithm Square-and-Multiply modulo $b(x) \in F[x]$

Input: $m \in \mathbb{N}$ and $a, b \in F[x]$ of degree $\deg a < d = \deg b$.
Output: $r = a^m \mod b$.

> **set** $r = a$ and **let** $m = m_l \cdots m_2 m_1$ in binary.
> **for** $k = l - 1$ **downto** 1 **do**
>   **set** $s = r^2$ .... $1M(d)$
>   **set** $r = s \mod b$ .... $4M(d)$
>   **if** $m_k = 1$ **set** $r = ar \mod b$ ..... $(a = x + \alpha)$ ..... $O(d)$
> **return** r

Costs $\mathbf{5}M(d)$ per iteration.
MCA: $\mathbf{3}M(d)$ by precomputing b_r^{-1}.
MCA: $\mathbf{2}M(d)$ by precomputing $FFT_\omega(b_r^{-1})$ and $FFT_\omega(b_r)$.
MBM: $\mathbf{1}M(d)$ by staying in FFT co-ordinates.

Algorithm Square-and-Multiply modulo $b(x) \in F[x]$

Input: $m \in \mathbb{N}$ and $a, b \in F[x]$ of degree $\deg a < d = \deg b$.
Output: $r = a^m \mod b$.

> **set** $r = a$ and **let** $m = m_l \cdots m_2 m_1$ in binary.
> **for** $k = l - 1$ **downto** 1 **do**
>   **set** $s = r^2$ .... $1M(d)$
>   **set** $r = s \mod b$ .... $4M(d)$
>   **if** $m_k = 1$ **set** $r = ar \mod b$ ..... $(a = x + \alpha)$ ..... $O(d)$
> **return** r

Costs $\mathbf{5}M(d)$ per iteration.
MCA: $\mathbf{3}M(d)$ by precomputing b_r^{-1}.
MCA: $\mathbf{2}M(d)$ by precomputing $FFT_\omega(b_r^{-1})$ and $FFT_\omega(b_r)$.
MBM: $\mathbf{1}M(d)$ by staying in FFT co-ordinates.

First idea: precompute $FFT_\omega(b_r^{-1})$ and $FFT_\omega(b_r)$

set $s = r^2$ .... 2 FFTs
set $d_q = 2d_r - d$.
if $d_q \geq 0$ **then** compute $r = s \mod b$:
  set $t = \lfloor s \rfloor_{d_q}$ .... $O(d)$
  set $q_r = t_r \cdot b_r^{-1}$ .... 2 FFTs
  set $q_r = \lfloor q_r \rfloor_{d_q}$ .... $O(d)$
  set $r_r = s_r - b_r q_r$ .... 2 FFTs

$$r_r = [\underbrace{0, 0, \cdots, 0}_{dq+1 \text{ zeroes}}, \underbrace{\square, \square, \cdots, \square}_{\text{remainder}}]$$

  set $r_r = r_r / x^{dq+1}$ .... $O(d)$
  set $d_r = \deg r$ .... $O(d)$

We have 6 FFTs of degree $< 2d \equiv \mathbf{2}M(d)$.

Main idea: stay in FFT co-ordinates

set $s_r = r_r^2$ .... $O(d)$
set $d_q = 2d_r - d$.
if $d_q \geq 0$ **then** compute $r = s \mod b$:
  set $t_r = \lfloor s_r \rfloor_{d_q}$ .... 2 FFTs
  set $q_r = t_r \cdot b_r^{-1}$ .... $O(d)$
  set $q_r = \lfloor q_r \rfloor_{d_q}$ .... 2 FFTs
  set $r_r = s_r - b_r q_r$ .... $O(d)$

$$r_r = [\underbrace{0, 0, \cdots, 0}_{dq+1 \text{ zeroes}}, \underbrace{\square, \square, \cdots, \square}_{\text{remainder}}]$$

  set $r_r = r_r / x^{dq+1}$ .... $O(d)$
  set $d_r = \deg r$???

We have 4 FFTs of degree $< 2d \equiv \frac{4}{3} M(d)$.

Main idea: stay in FFT co-ordinates

set $s_r = r_r^2$ .... $O(d)$

set $d_q = 2d_r - d$.

if $d_q \geq 0$ **then** compute $r = s \mod b$:

set $t_r = \lfloor s_r \rfloor_{d_q}$ ..... $s_r = [\underbrace{0, 0}_{\delta \text{ zeroes}}\ \square, \cdots, \square]$ ..... 2 FFTs

if $\delta > 0$ **set** $d_q = d_q - \delta$ and $s_r = s_r / x^{\delta}$ .... $O(d)$

set $q_r = t_r \cdot b_r^{-1}$ .... $O(d)$

set $q_r = \lfloor q_r \rfloor_{d_q}$ .... 2 FFTs

set $r_r = s_r - b_r q_r$ .... $O(d)$

set $r_r = r_r / x^{dq+1}$ .... $O(d)$

$$r_r = [\underbrace{0, \square, \cdots, \square}_{\textbf{remainder of degree d}-\textbf{2}}, \square, \cdots, \square]$$

set $d_r = d - 1$

Final idea: do 2 larger FFTs

set $s_r = r_r^2$ .... $O(d)$
set $d_q = 2d_r - d$.
if $d_q \geq 0$ **then** compute $r = s \mod b$:
// **set** $t_r = \lfloor s_r \rfloor_{d_q}$ .... $s_r = [\underbrace{0,0}_{\delta \text{ zeroes}} \square, \cdots, \square]$ .... OMIT
set $q_r = s_r \cdot b_r^{-1}$ ..... $q_r = [\underbrace{0,0}_{\delta \text{ zeroes}} \square, \cdots, \square]$ ..... $O(d)$
set $q_r = \lfloor q_r \rfloor_{d_q}$ (has degree $< 3d$) ....2 FFTs
if $\delta > 0$ **set** $d_q = d_q - \delta$ and $s_r = s_r / x^\delta$
set $r_r = s_r - b_r q_r$ .... $O(d)$
set $r_r = r_r / x^{dq+1}$ .... $O(d)$
set $d_r = d - 1$

We have 2 FFTs of degree $< 3d \equiv 1M(d)$.

A benchmark

Compute the $d-3$ roots of $f(x) = (x^d - 1)/(x^2 - 1)$ in $\mathbb{F}_p$ for $d = 2^k$ where $p = 2^{20}1017 + 1$.

				New		
d	Maple 14	Mahdi	Magma	Classical	FFT	Lehmer GCD
4096	24.0s	9.2s	7.0s	2.55s	0.8s	0.58s
8192	96.3s	34.7s	17.2s	10.4s	2.3s	1.50s
16384	339.7s		48.9s	39.4s	7.2s	4.2s

Maple is using classical polynomial arithmetic $O(\log(p) d^2 \log p)$.
Magma is using fast polynomial arithmetic $O(d \log^2 d \log p)$.

Current and Future Work

- fast Euclidean algorithm for GCD [Soo Go]
- parallelize the 4 multiplications inside the fast Euclidean algorithm
- need parallel FFT for large d
- after splitting $f(x)$ compute the roots recursively in parallel

Appendix: Maple and Magma code

Maple 14 code	Magma code
`> p := 2114977793;`	`> p := 2114977793;`
	`> Fp := GaloisField(p);`
	`> Zpx<x> := PolynomialRing(Fp);`
`> d := 8192;`	`> d := 8192;`
`> divide(x^d-1,x^2-1,'f');`	`> f := ExactQuotient(x^d-1,x^2-1);`
`> nops(Roots(f) mod p);`	`> #Roots(f);`
`> quit;`	`> quit;`