Computing one billion roots using the tangent Graeffe method *

Joris van der Hoeven
CNRS, École polytechnique, Institut Polytechnique de Paris
Laboratoire d'informatique de l'École polytechnique (LIX, UMR 7161)
1, rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing, CS35003
91120 Palaiseau, France
Email: vdhoeven@lix.polytechnique.fr

Michael Monagan
Department of Mathematics
Simon Fraser University
8888 University Drive
Burnaby, British Columbia
V5A 1S6, Canada
Email: mmonagan@cecm.sfu.ca

February 4, 2021

Abstract

Let p be a prime of the form $p = \sigma 2^k + 1$ with σ small and let $\mathbb{F}_p$ denote the finite field with p elements. Let $P(z)$ be a polynomial of degree d in $\mathbb{F}_p[z]$ with d distinct roots in $\mathbb{F}_p$. For $p = 5 \cdot 2^{55} + 1$ we can compute the roots of such polynomials of degree 10^9. We believe we are the first to factor such polynomials of size one billion. We used a multi-core computer with two 10 core Intel Xeon E5 2680 v2 CPUs and 128 gigabytes of RAM. The factorization takes just under 4,000 seconds on 10 cores and uses 121 gigabytes of RAM.

We used the tangent Graeffe root finding algorithm from [27, 19] which is a factor of $O(\log d)$ faster than the Cantor–Zassenhaus algorithm. We implemented the tangent Graeffe algorithm in C using our own library of 64 bit integer FFT based in-place polynomial algorithms then parallelized the FFT and main steps using Cilk C.

In this article we discuss the steps of the tangent Graeffe algorithm, the sub-algorithms that we used, how we parallelized them, and how we organized the memory so we could factor a polynomial of degree 10^9. We give both a theoretical and practical comparison of the tangent Graeffe algorithm with the Cantor–Zassenhaus algorithm for root finding. We improve the complexity of the tangent Graeffe algorithm by a factor of 2. We present a new in-place product tree multiplication algorithm that is fully parallelizable. We present some timings comparing our software with Magma's polynomial factorization command.

Polynomial root finding over smooth finite fields is a key ingredient for algorithms for sparse polynomial interpolation that are based on geometric sequences. This application was also one of our main motivations for the present work.

1 Introduction

Consider a polynomial function $f : \mathbb{K}^n \to \mathbb{K}$ over a field $\mathbb{K}$ given through a black box capable of evaluating f at points in $\mathbb{K}^n$. The problem of *sparse interpolation* is to recover the representation of $f \in \mathbb{K}[x_1, \ldots, x_n]$ in its usual

Note: This work was undertaken while the first author was on study leave at Simon Fraser University in 2020 during the COVID pandemic. It is part of projects that received funding from the French "Agence de l'innovation de défense" and the "National Science and Engineering Research Council of Canada".

form, as a linear combination

$$f = \sum_{1 \leqslant i \leqslant t} c_i \boldsymbol{x}^{\boldsymbol{e}_i} \tag{1}$$

of monomials $\boldsymbol{x}^{\boldsymbol{e}_i} = x_1^{e_{1,1}} \cdots x_n^{e_{1,n}}$. One popular approach to sparse interpolation is to evaluate f at points in a geometric progression. This approach goes back to work of Prony in the eighteen's century [41] and became well known after Ben-Or and Tiwari's seminal paper [2]. It has widely been used in computer algebra, both in theory [35, 7, 34, 39, 30, 32, 33] and in practice [14, 11, 31, 25, 29, 24]; see [43] for a nice survey.

More precisely, if a bound T for the number of terms t is known, then we first evaluate f at $2T$ pairwise distinct points $\boldsymbol{\alpha}^0, \boldsymbol{\alpha}^1, \ldots, \boldsymbol{\alpha}^{2T-1}$, where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n) \in \mathbb{K}^n$ and $\boldsymbol{\alpha}^k := (\alpha_1^k, \ldots, \alpha_n^k)$ for all $k \in \mathbb{N}$. The generating function of the evaluations at $\boldsymbol{\alpha}^k$ satisfies the identity

$$\sum_{k \in \mathbb{N}} f(\boldsymbol{\alpha}^k) z^k = \sum_{1 \leqslant i \leqslant t} \sum_{k \in \mathbb{N}} c_i \boldsymbol{\alpha}^{\boldsymbol{e}_i k} z^k = \sum_{1 \leqslant i \leqslant t} \frac{c_i}{1 - \boldsymbol{\alpha}^{\boldsymbol{e}_i} z} = \frac{N(z)}{\Lambda(z)},$$

where $\Lambda = (1 - \boldsymbol{\alpha}^{\boldsymbol{e}_1} z) \cdots (1 - \boldsymbol{\alpha}^{\boldsymbol{e}_t} z)$ and $N \in \mathbb{K}[z]$ is of degree $< t$. The rational function N/Λ can be recovered from $f(\boldsymbol{\alpha}^0), f(\boldsymbol{\alpha}^1), \ldots, f(\boldsymbol{\alpha}^{2T-1})$ using fast Padé approximation [6, 37]. For well chosen points $\boldsymbol{\alpha}$, it is often possible to recover the exponents $\boldsymbol{e}_i$ from the values $\boldsymbol{\alpha}^{\boldsymbol{e}_i} \in \mathbb{K}$. If the exponents $\boldsymbol{e}_i$ are known, then the coefficients c_i can also be recovered using a transposed version of fast multipoint interpolation [7, 5]. This leaves us with the question how to compute the roots $\boldsymbol{\alpha}^{-\boldsymbol{e}_i}$ of Λ in an efficient way.

For practical applications in computer algebra, we usually have $\mathbb{K} = \mathbb{Q}$, in which case it is most efficient to use a multi-modular strategy. This means that we rather work with coefficients in a finite field $\mathbb{K} = \mathbb{F}_p$, where p is a prime number that we are free to choose. It is well known that polynomial arithmetic over $\mathbb{F}_p$ can be implemented most efficiently using FFTs when the order $p - 1$ of the multiplicative group is smooth. In practice, this prompts us to choose p of the form $s2^l + 1$ for some small s and such that p fits into a machine word.

The traditional way to compute roots of polynomials over finite fields is using Cantor and Zassenhaus' method [8]. In [19, 20], alternative algorithms were proposed for our case of interest when $p - 1$ is smooth. The fastest algorithm was based on the *tangent Graeffe transform* and it gains a factor $\log t$ with respect to Cantor–Zassenhaus' method. The aim of the present paper is to report on a parallel implementation of this new algorithm and on a few improvements that allow for a further constant speed-up. A preliminary version of this paper was published at ICMS 2020 [27]. The new contributions in this paper include the theoretical contributions in sections 3.4, 3.5, 3.6 and 3.7, the implementation details in section 4 and the (parallel) timing results in section 5.

In section 2, we start by recalling generalities about the Graeffe transform and the heuristic root finding method based on the tangent Graeffe transform from [19]. In section 3, we present the main new theoretical improvements, which all rely on optimizations in the FFT-model for fast polynomial arithmetic. Our contributions are threefold:

- In the FFT-model, one backward transform out of four can be saved for Graeffe transforms of order two (see section 3.2).
- When composing a large number of Graeffe transforms of order two, "FFT caching" [3] can be used to gain another factor of 3/2 (see section 3.3).
- All optimizations still apply in the TFT model, which can save a factor between one and two, depending on the number of roots (see section 3.5).

In section 3.4 we also indicate how to generalize our methods to Graeffe transforms of general orders. In section 3.7 we determine how much faster the tangent Graeffe algorithm is than the Cantor–Zassenhaus algorithm. To do this, we determine the constant factors in the complexities of both algorithms, under the assumption that arithmetic in $\mathbb{F}_p[x]$ is done using FFT-based algorithms.

We first implemented a sequential version of the tangent Graeffe method in C, with the optimizations from sections 3.2 and 3.3; see [27]. Section 4 is devoted to a more elaborate parallel implementation in Cilk C. We detail how we parallelized the different steps of the algorithm, and how we organized the memory so we could factor a polynomial of degree 10^9 over $\mathbb{F}_p$, for $p = 5 \cdot 2^{55} + 1$. We believe we are the first to factor such large polynomials. Our code is available here

`http://www.cecm.sfu.ca/CAG/code/TangentGraeffe`

In the last section 5, we give timings. We used a multi-core computer with two 10 core Intel Xeon E5 2680 v2 CPUs and 128 gigabytes of RAM. The above factorization of a polynomial of degree 10^9 then takes just under 4,000 seconds on 10 cores and uses 121 gigabytes of RAM.

2 Root finding using the tangent Graeffe transform

Let $\mathbb{K}$ be an effective field. Throughout the paper, time complexities count arithmetic operations in the field $\mathbb{K}$ and space complexities are for elements of $\mathbb{K}$. We use $\mathsf{M}(d)$ to denote the time for multiplying two polynomials in $\mathbb{K}[z]$ of degree $< d$. We make the customary assumption that $\mathsf{M}(n)/n$ is a non-decreasing function that tends to infinity.

We will chiefly be interested in the case when $\mathbb{K} = \mathbb{F}_p$, where p is a prime of the form $p = \sigma 2^k + 1$ with σ small. We call primes of this form *smooth Fourier primes*. Some examples that we use and their bit lengths are as follows:

p	$7 \cdot 2^{26} + 1$	$3 \cdot 2^{30} + 1$	$5 \cdot 2^{55} + 1$	$3 \cdot 29 \cdot 2^{56} + 1$	$5 \cdot 101 \cdot 2^{54} + 1$
$\log_2 p$	28.8	31.6	57.3	62.4	62.98

For $\mathbb{K} = \mathbb{F}_p$ and $p = \sigma 2^k + 1$ of this form and $2^k > 2d$, we have $\mathsf{M}(d) = O(d \log d)$, by using FFT-multiplication.

Let $P(z)$ be a polynomial of degree d in $\mathbb{F}_p[z]$ which has d distinct roots in $\mathbb{F}_p$. The tangent Graeffe algorithm computes the roots of $P(z)$. The cost of the algorithm depends on a parameter $s = \sigma 2^j$ with $0 \leqslant j \leqslant k$. The parameter s determines what proportion of the roots are found in each iteration of the algorithm. The space complexity of the algorithm is $\Theta(s + d)$ and the average time complexity is

$$O\left(\mathsf{M}(d) \log\left(\tfrac{p}{s}\right) + \mathsf{M}(s) + \mathsf{M}(d) \log d\right).$$

Theoretically, choosing $s \asymp d\sqrt{\log p}$ yields the best time complexity. However, because we want to factor polynomials with very large d, our implementation chooses a smaller s in the interval $[2d, 4d)$ to save space. For $s \in [2d, 4d)$ the time complexity is

$$O\left(\mathsf{M}(d) \log\left(\tfrac{p}{d}\right) + \mathsf{M}(d) + \mathsf{M}(d) \log d\right) = O(\mathsf{M}(d) \log p).$$

In this section, we recall the tangent Graeffe algorithm from [19]. In the next section, we will analyze its complexity in the FFT-model and present several improvements.

2.1 Graeffe transforms

Let $\mathbb{K}$ be a general field. The traditional *Graeffe transform* of a monic polynomial $P \in \mathbb{K}[z]$ of degree d is the unique polynomial $G(P) \in \mathbb{K}[z]$ of degree d such that

$$G(P)(z^2) = (-1)^d P(z) P(-z). \tag{2}$$

If P is monic, then so is $G(P)$. If P splits over $\mathbb{K}$ into linear factors $P = (z - \beta_1) \cdots (z - \beta_d)$, then one has

$$G(P) = (z - \beta_1^2) \cdots (z - \beta_d^2).$$

More generally, given $r \geqslant 2$, we define the *Graeffe transform of order r* to be the polynomial $G_r(P) \in \mathbb{K}[z]$ of degree d given by

$$G_r(P)(z) = \mathrm{Res}_u(P(u), z - u^r)$$

If $P = (z - \beta_1) \cdots (z - \beta_d)$, then

$$G_r(P) = (z - \beta_1^r) \cdots (z - \beta_d^r).$$

If ω is a primitive r-th root of unity in $\mathbb{K}$, then we also have

$$G_r(P)(z^r) = (-1)^{(r-1)d} P(z) P(\omega z) \cdots P(\omega^{r-1} z). \tag{3}$$

If $r, s \geqslant 2$, then we finally have

$$G_{rs} = G_r \circ G_s = G_s \circ G_r. \tag{4}$$

2.2 Root finding using tangent Graeffe transforms

Let ϵ be a formal indeterminate with $\epsilon^2 = 0$. Elements in $\mathbb{K}[\epsilon]/(\epsilon^2)$ are called *tangent numbers*. They are of the form $a + b\epsilon$ with $a, b \in \mathbb{K}$ and basic arithmetic operations go as follows:

$$\begin{aligned}(a + b\epsilon) \pm (c + d\epsilon) &= (a \pm c) + (b \pm d)\epsilon \\ (a + b\epsilon)(c + d\epsilon) &= ac + (ad + bc)\epsilon\end{aligned}$$

Now let $P \in \mathbb{K}[z]$ be a monic polynomial of degree d that splits over $\mathbb{K}$:

$$P = (z - \alpha_1) \cdots (z - \alpha_d),$$

where $\alpha_1, \ldots, \alpha_d \in \mathbb{K}$ are pairwise distinct. Then the *tangent deformation* $\tilde{P}(z) := P(z + \varepsilon)$ satisfies

$$\tilde{P} = P + P'\epsilon = (z - (\alpha_1 - \epsilon)) \cdots (z - (\alpha_d - \epsilon)).$$

The definitions from the previous subsection readily extend to coefficients in $\mathbb{K}[\epsilon]$ instead of $\mathbb{K}$. Given $r \geqslant 2$, we call $G_r(\tilde{P})$ the *tangent Graeffe transform* of P of order r. We have

$$G_r(\tilde{P}) = (z - (\alpha_1 - \epsilon)^r) \cdots (z - (\alpha_d - \epsilon)^r),$$

where

$$(\alpha_k - \epsilon)^r = \alpha_k^r - r\alpha_k^{r-1}\epsilon, \qquad k = 1, \ldots, d.$$

Now assume that we have an efficient way to determine the roots $\alpha_1^r, \ldots, \alpha_d^r$ of $G_r(P)$. For some polynomial $T \in \mathbb{K}[z]$, we may decompose

$$G_r(\tilde{P}) = G_r(P) + T\epsilon$$

For any root α_k^r of $G_r(P)$, we then have

$$\begin{aligned}G_r(\tilde{P})(\alpha_k^r - r\alpha_k^{r-1}\epsilon) &= G_r(P)(\alpha_k^r) + (T(\alpha_k^r) - G_r(P)'(\alpha_k^r)r\alpha_k^{r-1})\epsilon \\ &= (T(\alpha_k^r) - G_r(P)'(\alpha_k^r)r\alpha_k^{r-1})\epsilon \\ &= 0.\end{aligned}$$

Whenever α_k^r happens to be a single root of $G_r(P)$, it follows that

$$r\alpha_k^{r-1} = \frac{T(\alpha_k^r)}{G_r(P)'(\alpha_k^r)}.$$

If $\alpha_k^r \neq 0$, this finally allows us to recover α_k as

$$\alpha_k = r\frac{\alpha_k^r}{r\alpha_k^{r-1}} = r\frac{\alpha_k^r G_r(P)'(\alpha_k^r)}{T(\alpha_k^r)}.$$

2.3 Heuristic root finding over smooth finite fields

Assume now that $\mathbb{K} = \mathbb{F}_p$ is a finite field, where p is a prime number of the form $p = \sigma 2^m + 1$ for some small σ. Assume also that $\eta \in \mathbb{F}_p$ is a primitive element of order $p - 1$ for the multiplicative group of $\mathbb{F}_p$.

Let $P = (z - \alpha_1) \cdots (z - \alpha_d) \in \mathbb{F}_p[z]$ be as in the previous subsection. The tangent Graeffe method can be used to efficiently compute those α_k of P for which α_k^r is a single root of $G_r(P)$. In order to guarantee that there are a sufficient number of such roots, we first replace $P(z)$ by $P(z + \tau)$ for a random shift $\tau \in \mathbb{F}_p$, and use the following heuristic:

H For any subset $\{\alpha_1, \ldots, \alpha_d\} \subseteq \mathbb{F}_p$ of cardinality d and any $r \leqslant (p - 1)/(4d)$, there exist at least $p/2$ elements $\tau \in \mathbb{F}_p$ such that $\{(\alpha_1 - \tau)^r, \ldots, (\alpha_d - \tau)^r\}$ contains at least $2d/3$ elements.

For a random shift $\tau \in \mathbb{F}_p$ and any $r \leqslant (p - 1)/(4d)$, the assumption ensures with probability at least $1/2$ that $G_r(P(z + \tau))$ has at least $d/3$ single roots.

Now take r to be the largest power of two such that $r \leqslant (p - 1)/(4d)$ and let $s = (p - 1)/r$. By construction, note that $s = O(d)$. The roots $\alpha_1^r, \ldots, \alpha_d^r$ of $G_r(P)$ are all s-th roots of unity in the set $\{1, \omega, \ldots, \omega^{s-1}\}$, where $\omega = \eta^r$. We may thus determine them by evaluating $G_r(P)$ at ω^i for $i = 0, \ldots, s - 1$. Since $s = O(d)$, this can be done efficiently using a discrete Fourier transform. Combined with the tangent Graeffe method from the previous subsection, this leads to the following probabilistic algorithm for root finding:

Algorithm 1
Input: $P \in \mathbb{F}_p[z]$ of degree d and only order one factors, $p = \sigma 2^m + 1$
Output: the set $\{\alpha_1, \ldots, \alpha_d\}$ of roots of P
Note: time complexities for the main steps are indicated on the right

1. If $d = 0$ then return $\varnothing$
2. Let $r = 2^N \in 2^{\mathbb{N}}$ be largest such that $r \leqslant (p-1)/(4d)$ and let $s := (p-1)/r$
3. Pick $\tau \in \mathbb{F}_p$ at random and compute $P^* := P(z+\tau) \in \mathbb{F}_p[z]$ ..... $O(\mathsf{M}(d))$
4. Compute $\tilde{P}(z) := P^*(z+\epsilon) = P^*(z) + P^*(z)'\epsilon \in (\mathbb{F}_p[\epsilon]/(\epsilon^2))[z]$
5. For $i = 1, \ldots, N$, set $\tilde{P} := G_2(\tilde{P}) \in (\mathbb{F}_p[\epsilon]/(\epsilon^2))[z]$ ..... $O\left(\mathsf{M}(d) \log\left(\frac{p}{s}\right)\right)$
6. Let $\omega \in \mathbb{F}_p^*$ be of order s and write $\tilde{P} = A + B\epsilon$
   Compute $A(\omega^i)$, $A'(\omega^i)$, and $B(\omega^i)$ for $i = 0, \ldots, s-1$ ..... $O(\mathsf{M}(s))$
7. If $P(\tau) = 0$, then set $S := \{\tau\}$, else set $S := \varnothing$
8. For $\beta \in \{1, \omega, \ldots, \omega^{s-1}\}$ do ..... $O(s)$
   If $A(\beta) = 0$ and $A'(\beta) \neq 0$, then set $S := S \cup \{r\beta A'(\beta)/B(\beta) + \tau\}$
9. Compute $Q := \prod_{\alpha \in S}(z - \alpha)$ ..... $O(\mathsf{M}(d) \log d)$
10. Compute $R := P/Q$ ..... $O(\mathsf{M}(d))$
11. Recursively determine the set of roots S' of R and return $S \cup S'$

Remark 1 To compute $G_2(\tilde{P}) = G_2(A + B\epsilon)$ we may use

$$(-1)^d G_2(\tilde{P}(z^2)) = A(z)A(-z) + (A(z)B(-z) + B(z)A(-z))\epsilon,$$

which requires three polynomial multiplications in $\mathbb{F}_p[z]$ of degree d. In total, step 5 therefore performs $O(N) = O(\log(p/s))$ such multiplications. We discuss how to perform step 5 efficiently in the FFT model in section 3.

Remark 2 For practical implementations, one may vary the threshold $r \leqslant (p-1)/(4d)$ for r and the resulting threshold $s \geqslant 4d$ for s. For larger values of s, the computations of the DFTs in step 6 get more expensive, but the proportion of single roots goes up, so more roots are determined at each iteration. From an asymptotic complexity perspective, it would be best to take $s \asymp d\sqrt{\log p}$. In practice, we actually preferred to take the *lower* threshold $s \geqslant 2d$, because the constant factor of our implementation of step 6 (based on Bluestein's algorithm [4]) is significant with respect to our highly optimized implementation of the tangent Graeffe method. A second reason we prefer s of size $O(d)$ instead of $O(d\sqrt{\log p})$ is that the total space used by the algorithm is linear in s.

Remark 3 For the application to sparse interpolation, it is possible to further speed up step 5 for the top-level iteration, which is the most expensive step. More precisely, for a polynomial with t terms, the idea is to take $\tau = 0$ and η of order $\approx t^c$ instead of $p-1$ for some constant c with $1 < c < 3$. This reduces $\log r$ (and the cost of the top-level iteration) by a factor of $\Theta(\log p / \log t)$. For the recursive calls, we still need to work with a primitive root of unity η' of order $p-1$ and random shifts.

3 Computing Graeffe transforms

3.1 Reminders about discrete Fourier transforms

Assume that $n \in \mathbb{N}$ is invertible in $\mathbb{K}$ and let $\omega \in \mathbb{K}$ be a primitive n-th root of unity. Consider a polynomial $A = a_0 + a_1 z + \cdots + a_{n-1} z^{n-1} \in \mathbb{K}[z]$. Then the discrete Fourier transform (DFT) of order n of the sequence $(a_i)_{0 \leqslant i < n}$ is defined by

$$\mathrm{DFT}_\omega((a_i)_{0 \leqslant i < n}) := (\hat{a}_k)_{0 \leqslant k < n}, \qquad \hat{a}_k := A(\omega^k).$$

We will write $\mathsf{F}(n)$ for the cost of one discrete Fourier transform in terms of the number of operations in $\mathbb{K}$. We assume that $n = o(\mathsf{F}(n))$. For any $i \in \{0, \ldots, n-1\}$, we have

$$\mathrm{DFT}_{\omega^{-1}}((\hat{a}_k)_{0\leqslant k<n})_i \quad = \quad \sum_{0\leqslant k<n} \hat{a}_k \omega^{-ik} \quad = \quad \sum_{0\leqslant j<n} a_j \sum_{0\leqslant k<n} \omega^{(j-i)k} \quad = \quad n a_i. \tag{5}$$

If n is invertible in $\mathbb{K}$, then it follows that $\mathrm{DFT}_\omega^{-1} = n^{-1}\,\mathrm{DFT}_{\omega^{-1}}$. The costs of direct and inverse transforms therefore coincide up to a factor $O(n)$.

If $n = n_1 n_2$ is composite, $0 \leqslant k_1 < n_1$, and $0 \leqslant k_2 < n_2$, then we have

$$\begin{aligned}
\hat{a}_{k_2 n_1 + k_1} &= \sum_{0\leqslant i_2<n_2} \sum_{0\leqslant i_1<n_1} a_{i_1 n_2 + i_2} \omega^{(i_1 n_2 + i_2)(k_2 n_1 + k_1)} \\
&= \sum_{0\leqslant i_2<n_2} \omega^{i_2 k_1} \left[\sum_{0\leqslant i_1<n_1} a_{i_1 n_2 + i_2} \omega^{i_1 n_2 k_1} \right] \omega^{i_2 k_2 n_1} \\
&= \sum_{0\leqslant i_2<n_2} \left[\omega^{i_2 k_1}\, \mathrm{DFT}_{\omega^{n_2}}((a_{i_1 n_2 + i_2})_{0\leqslant i_1<n_1})_{k_1}\right] \omega^{i_2 (k_2 n_1 + k_1)} \\
&= \mathrm{DFT}_{\omega^{n_1}} \left(\left(\omega^{i_2 k_1}\, \mathrm{DFT}_{\omega^{n_2}}((a_{i_1 n_2 + i_2})_{0\leqslant i_1<n_1})_{k_1}\right)_{0\leqslant i_2<n_2} \right)_{k_2}.
\end{aligned} \tag{6}$$

This shows that a DFT of length n reduces to n_1 transforms of length n_2 plus n_2 transforms of length n_1 plus n multiplications in $\mathbb{K}$:

$$\mathsf{F}(n_1 n_2) \leqslant n_1 \mathsf{F}(n_2) + n_2 \mathsf{F}(n_1) + O(n).$$

In particular, if $r = O(1)$, then $\mathsf{F}(rn) \sim r\mathsf{F}(n)$.

It is sometimes convenient to apply DFTs directly to polynomials as well; for this reason, we also define $\mathrm{DFT}_\omega(A) := (\hat{a}_k)_{0\leqslant k<n}$. Given two polynomials $A, B \in \mathbb{K}[z]$ with $\deg(AB) < n$, we may then compute the product AB using

$$AB \quad = \quad \mathrm{DFT}_\omega^{-1}(\mathrm{DFT}_\omega(A)\,\mathrm{DFT}_\omega(B)).$$

In particular, we obtain $\mathsf{M}(n) \sim 3\mathsf{F}(2n) \sim 6\mathsf{F}(n)$, where we recall that $\mathsf{M}(n)$ stands for the cost of multiplying two polynomials of degree $< n$.

Remark 4 In Algorithm 1, we note that step 6 comes down to the computation of three DFTs of length s. Since r is a power of two, this length is of the form $s = \sigma 2^k$ for some $k \in \mathbb{N}$. In view of (6), we may therefore reduce step 6 to 3σ DFTs of length 2^k plus $3 \cdot 2^k$ DFTs of length σ. If σ is very small, then we may use a naive implementation for DFTs of length σ. In general, one may use Bluestein's algorithm [4] to reduce the computation of a DFT of length σ into the computation of a product in $\mathbb{K}[z]/(z^\sigma - 1)$, which can in turn be computed using FFT-multiplication and three DFTs of length a larger power of two.

3.2 Graeffe transforms of order two

Let $\mathbb{K}$ be a field with a primitive $(2n)$-th root of unity ω. Let $P \in \mathbb{K}[z]$ be a polynomial of degree $d = \deg P < n$. Then the relation (2) yields

$$G(P)(z^2) = (-1)^d\, \mathrm{DFT}_\omega^{-1}(\mathrm{DFT}_\omega(P(z))\,\mathrm{DFT}_\omega(P(-z))). \tag{7}$$

For any $k \in \{0, \ldots, 2n-1\}$, we further note that

$$\mathrm{DFT}_\omega(P(-z))_k = P(-\omega^k) = P(\omega^{(k+n) \bmod 2n}) = \mathrm{DFT}_\omega(P(z))_{(k+n) \bmod 2n}\,, \tag{8}$$

so $\mathrm{DFT}_\omega(P(-z))$ can be obtained from $\mathrm{DFT}_\omega(P)$ using n transpositions of elements in $\mathbb{K}$. Concerning the inverse transform, we also note that

$$\mathrm{DFT}_\omega(G(P)(z^2))_k = G(P)(\omega^{2k}) = \mathrm{DFT}_{\omega^2}(G(P))_k,$$

for $k = 0, \ldots, n-1$. Plugging this into (7), we conclude that

$$G(P) = (-1)^d \operatorname{DFT}_{\omega^2}^{-1}((\operatorname{DFT}_\omega(P)_k \operatorname{DFT}_\omega(P)_{k+n})_{0 \leqslant k < n}).$$

This leads to the following algorithm for the computation of $G(P)$:

Algorithm 2
Input: $P \in \mathbb{K}[z]$ with $d = \deg P < n$ and a primitive $(2n)$-th root of unity $\omega \in \mathbb{K}$
Output: $G(P)$

---

1. Compute $(\hat{P}_k)_{0 \leqslant k < 2n} := \operatorname{DFT}_\omega(P)$
2. For $k = 0, \ldots, n-1$, compute $\hat{G}_k := (-1)^d \hat{P}_k \hat{P}_{k+n}$
3. Return $\operatorname{DFT}_{\omega^2}^{-1}((\hat{G}_k)_{0 \leqslant k < n})$

Proposition 1 *Let $\omega \in \mathbb{K}$ be a primitive $2n$-th root of unity in $\mathbb{K}$ and assume that 2 is invertible in $\mathbb{K}$. Given a monic polynomial $P \in \mathbb{K}[z]$ with $\deg P < n$, we can compute $G(P)$ in time*

$$\mathsf{G}_2(n) \sim 3\mathsf{F}(n).$$

Proof We have already explained the correctness of Algorithm 2. Step 1 requires one forward DFT of length $2n$ and cost $\mathsf{F}(2n) = 2\mathsf{F}(n) + O(n)$. Step 2 can be done in linear time $O(n)$. Step 3 requires one inverse DFT of length n and cost $\mathsf{F}(n) + O(n)$. The total cost of Algorithm 2 is therefore $3\mathsf{F}(n) + O(n) \sim 3\mathsf{F}(n)$. □

Remark 5 In terms of the complexity of multiplication, we obtain $\mathsf{G}_2(n) \sim 1/2\mathsf{M}(n)$. This gives a 33.3% improvement over the previously best known bound $\mathsf{G}_2(n) \sim 2/3\mathsf{M}(n)$ that was used in [19]. Note that the best known algorithm for computing squares of polynomials of degree $< n$ is $\sim 2/3\mathsf{M}(n)$. It would be interesting to know whether squares can also be computed in time $\sim 1/2\mathsf{M}(n)$.

3.3 Graeffe transforms of power of two orders

In view of (4), Graeffe transforms of power of two orders 2^m can be computed using

$$G_{2^m}(P) = \left(G \circ \overset{m}{\cdots} \circ G\right)(P). \tag{9}$$

Now assume that we computed the first Graeffe transform $G(P)$ using Algorithm 2 and that we wish to apply a second Graeffe transform to the result. Then we note that

$$\operatorname{DFT}_\omega(G(P))_{2k} = \operatorname{DFT}_{\omega^2}(G(P))_k = \hat{G}_k \tag{10}$$

is already known for $k = 0, \ldots, n-1$. We can use this to accelerate step 1 of the second application of Algorithm 2. Indeed, in view of (6) for $n_1 = 2$ and $n_2 = n$, we have

$$\operatorname{DFT}_\omega(G(P))_{2k+1} = \operatorname{DFT}_{\omega^2}\left(\left(\omega^i G(P)_i\right)_{0 \leqslant i < n}\right)_k \tag{11}$$

for $k = 0, \ldots, n-1$. In order to exploit this idea in a recursive fashion, it is useful to modify Algorithm 2 so as to include $\operatorname{DFT}_{\omega^2}(P)$ in the input and $\operatorname{DFT}_{\omega^2}(G(P))$ in the output. This leads to the following algorithm:

Algorithm 3
Input: $P \in \mathbb{K}[z]$ with $d = \deg P < n$, a primitive $(2n)$-th root of unity $\omega \in \mathbb{K}$,
and $(\hat{Q}_k)_{0 \leqslant k < n} = \operatorname{DFT}_{\omega^2}(P)$
Output: $G(P)$ and $\operatorname{DFT}_{\omega^2}(G(P))$

---

1. Set $(\hat{P}_{2k})_{0\leqslant k<n} := (\hat{Q}_k)_{0\leqslant k<n}$
2. Set $(\hat{P}_{2k+1})_{0\leqslant k<n} := \mathrm{DFT}_{\omega^2}((\omega^i P_i)_{0\leqslant i<n})$
3. For $k = 0, \ldots, n-1$, compute $\hat{G}_k := (-1)^d \hat{P}_k \hat{P}_{k+n}$
4. Return $\mathrm{DFT}_{\omega^2}^{-1}((\hat{G}_k)_{0\leqslant k<n})$ and $(\hat{G}_k)_{0\leqslant k<n}$

Proposition 2 *Let $\omega \in \mathbb{K}$ be a primitive $2n$-th root of unity in $\mathbb{K}$ and assume that 2 is invertible in $\mathbb{K}$. Given a monic polynomial $P \in \mathbb{K}[z]$ with $\deg P < n$ and $m \geqslant 1$, we can compute $G_{2^m}(P)$ in time*

$$\mathsf{G}_{2^m}(n) \sim (2m+1)\mathsf{F}(n).$$

Proof It suffices to compute $\mathrm{DFT}_{\omega^2}(P)$ and then to apply Algorithm 3 recursively, m times. Every application of Algorithm 3 now takes $2\mathsf{F}(n) + O(n) \sim 2\mathsf{F}(n)$ operations in $\mathbb{K}$, whence the claimed complexity bound. □

Remark 6 In [19], Graeffe transforms of order 2^m were directly computed using the formula (9), using $\sim 4m\mathsf{F}(n)$ operations in $\mathbb{K}$. The new algorithm is twice as fast for large m.

3.4 Graeffe transforms of arbitrary smooth orders

The algorithms from subsections 3.2 and 3.3 readily generalize to Graeffe transforms of order r^m for arbitrary $r \geqslant 2$, provided that we have an (rn)-th root of unity $\omega \in \mathbb{K}$. For convenience of the reader, we specified the generalization of Algorithm 3 below, together with the resulting complexity bounds.

Algorithm 4
Input: $P \in \mathbb{K}[z]$ with $\deg P < n$, $r \geqslant 2$, a primitive (rn)-th root of unity $\omega \in \mathbb{K}$, and $(\hat{Q}_k)_{0\leqslant k<n} = \mathrm{DFT}_{\omega^r}(P)$
Output: $G_r(P)$ and $\mathrm{DFT}_{\omega^r}(G_r(P))$

---

1. Set $(\hat{P}_{kr})_{0\leqslant k<n} := (\hat{Q}_k)_{0\leqslant k<n}$
2. For $j = 1, \ldots, r-1$, set $(\hat{P}_{kr+j})_{0\leqslant k<n} := \mathrm{DFT}_{\omega^r}((\omega^{ij} P_i)_{0\leqslant i<n})$
3. For $k = 0, \ldots, n-1$, compute $\hat{G}_k := (-1)^{(r-1)d} \hat{P}_k \hat{P}_{k+n} \cdots \hat{P}_{k+(r-1)n}$
4. Return $\mathrm{DFT}_{\omega^r}^{-1}((\hat{G}_k)_{0\leqslant k<n})$ and $(\hat{G}_k)_{0\leqslant k<n}$

Proposition 3 *Let $\omega \in \mathbb{K}$ be a primitive (rn)-th root of unity in $\mathbb{K}$, where $r \geqslant 2$ is invertible in $\mathbb{K}$. Given a monic polynomial $P \in \mathbb{K}[z]$ with $\deg P < n$ and $m \geqslant 1$, we can compute $G_{r^m}(P)$ in time*

$$\mathsf{G}_{r^m}(n) \sim (rm+1)\mathsf{F}(n).$$

Proof Straightforward generalization of Proposition 2. □

Corollary 1 *Let $\omega \in \mathbb{K}$ be a primitive $(r_1 \cdots r_\tau n)$-th root of unity in $\mathbb{K}$, where $r_1 \geqslant 2, \ldots, r_\tau \geqslant 2$ are invertible in $\mathbb{K}$. Given a monic polynomial $P \in \mathbb{K}[z]$ with $\deg P < n$ and $m_1, \ldots, m_\tau \in \mathbb{N}$, we can compute $G_{r_1^{m_1} \cdots r_\tau^{m_\tau}}(P)$ in time*

$$\mathsf{G}_{r_1^{m_1} \cdots r_\tau^{m_\tau}}(n) \sim (r_1 m_1 + \cdots + r_\tau m_\tau + \tau)\mathsf{F}(n).$$

Proof Direct consequence of (4). □

Remark 7 In our application to root finding, we are interested in the efficient computation of Graeffe transforms of high order r^m. In terms of the size $\log r^m$ of r^m, it is instructive to observe that the "average cost"

$$\mathsf{A}_{r^m}(n) = \frac{\mathsf{G}_{r^m}(n)}{\log r^m \mathsf{F}(n)} \sim \frac{r}{\log r}$$

is minimal for $r = 3$. This suggests that it might be interesting to use Graeffe transforms of order three whenever possible. In the application of Algorithm 1, this would lead us to take primes of the form $p = \sigma \cdot 2^m \cdot 3^l + 1$, with σ small and $\sigma \cdot 2^m$ close to d. This still allows us to use radix 2 FFTs, while at the same time benefiting from radix 3 Graeffe transforms.

3.5 Truncated Fourier transforms

If $\mathbb{K} = \mathbb{F}_q$ is a fixed finite field, then DFTs are most efficient for sizes n that divide $q - 1$. For our root finding application, it is often convenient to take $q = 3 \cdot 2^{30} + 1$, in which case n should be a power of two or three times a power of two. The truncated Fourier transform was developed for the multiplication of polynomials such that the degree of the product does not have a nice size n of this type. It turns out that we may also use it for the efficient computation of Graeffe transforms of polynomials of arbitrary degrees. Moreover, the optimizations from the previous subsections still apply.

Let us briefly describe how the truncated Fourier transform can be used for the computation of Graeffe transforms of power of two orders. With the notations from subsections 3.2 and 3.3, we assume that $2n = 2^\beta$ is a power of two as well and that we wish to compute the Graeffe transform of a polynomial P of degree $\deg P < t$ with $n/2 \leqslant t < n$. Let $[i]_\beta$ denote the reversal of a binary number $i \in \{0, \ldots, 2n-1\}$ of β bits. For instance, $[3]_4 = 12$ and $[5]_6 = 40$. Then the truncated Fourier of P at order $T \geqslant t$ is defined by

$$\mathrm{TFT}_{\omega,T}(P) := (P(\omega^{[0]_\beta}), P(\omega^{[1]_\beta}), \ldots, P(\omega^{[T-1]_\beta})).$$

It has been shown in [22] that $\tilde{P} := \mathrm{TFT}_{\omega,T}(P)$ and $P = \mathrm{TFT}^{-1}_{\omega,T}(\tilde{P})$ can both be computed in time $\sim (T/n)\mathsf{F}(n)$. More generally, for direct transforms, one may compute

$$\mathrm{TFT}_{\omega,\Delta,T}(P) := (P(\omega^{[\Delta]_\beta}), P(\omega^{[\Delta+1]_\beta}), \ldots, P(\omega^{[\Delta+T-1]_\beta}))$$

in time $\sim (T/n)\mathsf{F}(n)$, whenever $0 \leqslant \Delta < \Delta + T \leqslant n$. For generalizations to arbitrary radices, we refer to [36].

Taking $T = 2t$, we note that

$$P(\omega^{[2k+1]_\beta}) = P(\omega^{[2k]_\beta + n/2}) = P(-\omega^{[2k]_\beta})$$

for $k = 0, \ldots, t-1$. This provides us with the required counterpart of (8) for retrieving $\mathrm{TFT}_{\omega,2t}(P(-x))$ efficiently from $\mathrm{TFT}_{\omega,2t}(P)$. The relation (10) also has a natural counterpart:

$$\mathrm{TFT}_{\omega,2t}(G(P))_k = G(P)(\omega^{[k]_\beta}) = G(P)(\omega^{2[k]_{\beta-1}}) = \mathrm{TFT}_{\omega^2,t}(G(P))_k\,,$$

for $k = 0, \ldots, t-1$. This leads to the following refinement of Algorithm 3:

Algorithm 5
Input: $P \in \mathbb{K}[z]$ with $d = \deg P < t \leqslant n = 2^{\beta-1}$,
a primitive $(2n)$-th root of unity $\omega \in \mathbb{K}$, and $(\hat{Q}_k)_{0 \leqslant k < t} = \mathrm{TFT}_{\omega^2,t}(P)$
Output: $G(P)$ and $\mathrm{TFT}_{\omega^2,t}(G(P))$

---

1. Set $(\hat{P}_k)_{0 \leqslant k < t} := (\hat{Q}_k)_{0 \leqslant k < t}$
2. Set $(\hat{P}_{k+t})_{0 \leqslant k < t} := \mathrm{TFT}_{\omega,t,t}(P)$
3. For $k = 0, \ldots, t-1$, compute $\hat{G}_{2k} := (-1)^d \hat{P}_{2k} \hat{P}_{2k+1}$
4. Return $\mathrm{TFT}^{-1}_{\omega^2,t}((\hat{G}_{2k})_{0 \leqslant k < t})$ and $(\hat{G}_{2k})_{0 \leqslant k < t}$

Proposition 4 *Let $\omega \in \mathbb{K}$ be a primitive $2n$-th root of unity in $\mathbb{K}$, where $2n = 2^\beta$, and assume that 2 is invertible in $\mathbb{K}$. Given a monic polynomial $P \in \mathbb{K}[z]$ with $n/2 \leqslant \deg P < t \leqslant n$ and $m \geqslant 1$, we can compute $G_{2^m}(P)$ in time*

$$\mathsf{G}_{2^m}(t;n) \sim \frac{t}{n}(2m+1)\mathsf{F}(n).$$

Proof Straightforward adaptation of the proof of Proposition 2, while using [22]. □

3.6 Taylor shifts

In step 3 of Algorithm 1, we still need an algorithm to compute the Taylor shift $P(z+\tau)$. If the characteristic of $\mathbb{K}$ exceeds d, then it is (not so) well known [1, Lemma 3] that this can be done in time $\mathsf{M}(d)+O(n)$, using the following reduction to a single polynomial multiplication of degree d:

Algorithm 6
Input: $P \in \mathbb{K}[z]$ of degree $d < \operatorname{char}\mathbb{K}$ and $\tau \in \mathbb{K}$
Output: $P(z+\tau)$

---

1. $L := 0!P_0 + 1!P_1 z + \cdots + d!P_d z^d$
2. $\tilde{L} := z^d L(1/z)$
3. $E := 1 + \tau z + \frac{1}{2!}\tau^2 z^2 + \cdots + \frac{1}{d!}\tau^d z^d$
4. $\tilde{\Pi} := \tilde{L}E \operatorname{rem} z^{d+1}$
5. $\Pi := z^d \tilde{\Pi}(1/z)$
6. Return $\frac{1}{0!}\Pi_0 + \frac{1}{1!}\Pi_1 z + \cdots + \frac{1}{d!}\Pi_d z^d$

It is interesting to observe that Taylor shifts can still be computed in time $O(\mathsf{M}(d))$ in small characteristic, as follows:

Algorithm 7
Input: $P \in \mathbb{K}[z]$ of degree $d \geqslant p = \operatorname{char}\mathbb{K} > 0$ and $\tau \in \mathbb{K}$
Output: $P(z+\tau)$

---

1. Define $z_i = z^{p^i}$ for $i = 0, \ldots, k$ where $k = \lfloor \log d / \log p \rfloor$
2. Rewrite $P = \hat{P}(z_0, \ldots, z_k) \in \mathbb{K}[z_0, \ldots, z_k]$ with $\deg_{z_i} P < p$ for $i = 0, \ldots, k-1$
3. For $i = 0, \ldots, k$, replace $\hat{P} := \hat{P}(z_0, \ldots, z_{i-1}, z_i + \tau^{p^i}, z_{i+1}, \ldots, z_k)$
4. Return $\hat{P}(z, z^p, \ldots, z^{p^k})$

3.7 Comparison with Cantor–Zassenhaus' algorithm

We give a theoretical comparison of Algorithm 1 with the Cantor–Zassenhaus algorithm [8], where both algorithms have been optimized in the "FFT model" [44]. For this comparison, it is convenient to replace the "worst case" heuristic **H** by a more empirical assumption. More precisely, if we take $s \geqslant \lambda d$ for $\lambda \geqslant 1$, then the expected proportion of single roots is $\mathrm{e}^{-1/\lambda}$ (see e.g. [23]). This expected proportion is indeed observed in practice: see Table 3. In Algorithm 1, we took $\lambda = 4$.

Proposition 5 *Consider Algorithm 1 with the modification that we take $s \geqslant \lambda d$ instead of $s \geqslant 4d$ for some fixed $\lambda \geqslant 1$. Then the expected cost of Algorithm 1 is bounded by*

$$\left(\tfrac{1}{3}\mathrm{e}^{1/\lambda}\log_2\left(\tfrac{p}{s}\right) + \tfrac{1}{4}\log_2 d + O(1)\right)\mathsf{M}(d). \tag{12}$$

Proof We first analyze the cost of step 9. Let $\mathsf{T}(k)$ be the cost of the polynomial multiplications in the product tree algorithm, where k is the size of S. Exercise 10.3 of [17] shows that $\mathsf{T}(k) < 1/2\mathsf{M}(k)\log k + O(k\log k)$. The recurrence for $\mathsf{T}(k)$ is $\mathsf{T}(k) = 2\mathsf{T}(k/2) + \mathsf{M}(k/2)$ for $k > 1$ and $\mathsf{T}(1) = 0$. Solving this recurrence while using the assumption that $\mathsf{M}(k)/(k\log k)$ is non-decreasing yields $\mathsf{T}(k) = 1/4\mathsf{M}(k)\log_2 k + O(\mathsf{M}(k))$. Now let $k_1, \ldots, k_\ell$ be the successive sizes of S for the recursive calls of the algorithm, so that $k_1 + \cdots + k_\ell = d$. Thus the cost of all executions of step 9 is $\mathsf{T}(k_1) + \cdots + \mathsf{T}(k_\ell)$. Using again that $\mathsf{M}(k)/(k\log k)$ is non-decreasing, we have $\mathsf{T}(k_1) + \cdots + \mathsf{T}(k_\ell) \leqslant \mathsf{T}(d) = 1/4\mathsf{M}(d)\log_2 d + O(\mathsf{M}(d))$.

Let us next analyze the cost of the other steps until step 10 inclusive, but without the recursive calls in step 11. Set $N := \log_2((p-1)/s)$.

- To compute $P(z+\tau)$ in step 3 costs $O(\mathsf{M}(d))$; using [1, Lemma 3].
- Step 4 takes $O(d)$ to compute $(P^*)'(z)$.
- By Proposition 2, step 5 costs $(2N+1)\mathsf{F}(d) + O(d)$, which is equivalent to $1/3N\mathsf{M}(d)$.
- Step 6 is $O(\mathsf{M}(s)) = O(\mathsf{M}(d))$ using [4].
- The division P/Q in step 10 is $O(\mathsf{M}(d))$ using fast division [17].

Altogether, the cost of these steps is $\left(1/3\log_2\left(\frac{p}{s}\right) + O(1)\right)\mathsf{M}(d)$.

We now need to account for the cost of the recursive calls in step 11. For $s \geqslant \lambda d$ the tangent Graeffe algorithm will, under our hypothesis, on average, obtain at least $\mathrm{e}^{-1/\lambda}$ of the roots in each recursive call, leaving slightly less than $\varepsilon := 1 - \mathrm{e}^{-1/\lambda}$ of them to be found. Since $\sum_{i=0}^{\infty}\varepsilon^i = \frac{1}{1-\varepsilon} = \mathrm{e}^{1/\lambda}$ and $\mathsf{M}(d)\log d$ is non-decreasing, the total cost of Algorithm 1 is therefore as claimed. □

Remark 8 In the asymptotic region where $\log d = o(\log p)$, the bound (12) reduces to $\left(1/3\mathrm{e}^{1/\lambda} + o(1)\right)\log_2\left(\frac{p}{s}\right)\mathsf{M}(d)$. Since we may pick λ as large as we want, the complexity of Algorithm 1 is then bounded by $(1/3+\epsilon)\log_2\left(\frac{p}{2d}\right)\mathsf{M}(d)$ for any $\epsilon > 0$.

Assume from now on that we are in the asymptotic region where $\log d = o(\log p)$. Then Remark 8 shows that the cost of the tangent Graeffe method is $(1/3+\epsilon)\log_2\left(\frac{p}{2d}\right)\mathsf{M}(d)$ for any $\epsilon > 0$. The bottleneck of Cantor–Zassenhaus' algorithm in this region is modular exponentiation, which accounts for a total cost of $O(\mathsf{M}(d)\log p\log d)$ [17]. We wish to determine the constant factor in order to give an accurate comparison between the two algorithms. Given polynomials $S, P \in \mathbb{F}_p[z]$ with $\deg S < \deg P = d$, one modular exponentiation does $\log_2(p/2)$ modular squarings

$$R := S^2 \operatorname{rem} P$$

in a loop. However, in the first few iterations, the degree of S^2 is less then P so no division is needed. The number of steps with divisions is $\log_2(p/(2d))$. Using fast division [17], the remainder R can be computed with two multiplications of size d assuming the required inverse of $\tilde{P}$, the reciprocal of P, is precomputed. In the FFT model, one forward transform of S may be saved and the forward transform of the inverse of $\tilde{P}$ may be cached [44]. This costs $7\mathsf{F}(n) + O(n)$, i.e. at least $\sim 7/3\mathsf{M}(d) + O(d)$, since $n \geqslant 2d-1$. The cost of the top-level modular exponentiation is therefore equivalent to $7/3\mathsf{M}(d)\log_2(p/(2d))$. Using a similar recursion as for T in the proof of Proposition 5, the total cost of all modular compositions in CZ is equivalent to $7/6\mathsf{M}(d)\log_2(p/(2d))\log_2 d$ (strictly speaking, this is really an upper bound; but we indeed obtain an equivalence for common values of $\mathsf{M}(d)$, such as $cd\log d$ or $cd\log d\log\log d$). Altogether, this means that the tangent Graeffe algorithm is about $7/2\log_2 t$ times faster than Cantor–Zassenhaus.

Remark 9 In practice, we often have $\log p \asymp \log d$, and the complexity of Cantor–Zassenhaus also involves another $O(\mathsf{M}(d)\log^2 d)$ term due to gcd computations. The corresponding constant factor is fairly high, which makes the tangent Graeffe algorithm even more favorable in this asymptotic region.

4 Implementing the tangent Graeffe algorithm

For our implementation, we have been pursuing three goals. We first wanted to reimplement the tangent Graeffe algorithm from scratch using FFT-based polynomial arithmetic and compare it with a good implementation of the Cantor–Zassenhaus algorithm. Second, we wanted to factor a polynomial of degree 10^9. For this, it is important to consider space efficiency. Third, we wanted to parallelize the tangent Graeffe algorithm for a multi-core computer. Unlike the Cantor–Zassenhaus algorithm, the tangent Graeffe algorithm is gcd-free which makes it easier to parallelize. We first tried parallelizing only the FFTs that appear in steps 3, 5, 6, 9, and 10 of Algorithm 1 to see if this alone is sufficient to obtain good parallel speedup. It isn't.

Our main practical challenge is, however, space, not time. For a 64 bit prime p, and an input polynomial of degree 10^9, the input polynomial and output roots alone need 16 gigabytes of storage. Our first implementation exceeded the virtual memory of our machine which has 128 gigabytes of RAM plus 240 gigabytes of SSD swap space.

To reduce space we use in-place algorithms. In-place algorithms do not allocate new space (memory). They work in the input and output space and we allow them to be given additional working space. To parallelize a recursive in-place algorithm, we must partition the input, output and the temporary memory (if any) for recursive parallel calls. Our design is such that the space used does not increase with more cores. We refer the reader to Giorgi, Grene and Roche [18] for examples of in-place serial algorithms in computer algebra and a bibliography.

4.1 Parallelizing the FFTs

Number theoretic FFTs over $\mathbb{F}_p$ were first introduced by Pollard [40]. Let us first explain how we parallelized such FFTs using Cilk C. The strategy is classical, but it is convenient to detail it here, since we use the same strategy for parallelizing the other steps in Algorithm 1. Cilk [16] was designed to support the parallelization of recursive divide and conquer algorithms like the FFT for multi-core computers. To parallelize recursive algorithms using Cilk we first modify the code so that the memory locations that are updated in the recursive calls do not overlap.

Figure 1 shows our sequential C code for two radix 2 FFTs that we use. In the vernacular `fft1` is known as the "decimation-in-time" FFT and `fft2` as the "decimation-in-frequency" FFT. We refer the reader to Chu and George [9] for a discussion of the two FFTs. We note that Geddes, Czapor, and Labahn present only `fft1` in their book on computer algebra [10], whereas von zur Gathen and Gerhard present only `fft2` [17].

In the code in Figure 1, `LONG` is a macro for `long long int`, a 64 bit signed integer. The C functions `addmod`, `submod`, `mulmod` implement $+, -, \times$ in $\mathbb{F}_p$ respectively for $1 < p < 2^{63}$. For multiplication in $\mathbb{F}_p$ we use Roman Pearce's implementation of Möller and Granlund [38]. The input array A of size n is the main input to and output from the FFT. The input W is an array of size n containing powers of ω, a primitive n-th root of unity in $\mathbb{F}_p$. We precompute

$$W = [\,1, \omega, \omega^2, \ldots, \omega^{\frac{n}{2}-1}, \quad 1, \omega^2, \omega^4, \ldots, \omega^{\frac{n}{2}-2}, \quad 1, \omega^4, \omega^8, \ldots, \omega^{\frac{n}{2}-4}, \quad \ldots, 1, 0\,].$$

Precomputing W saves asymptotically half of the multiplications in the FFT. For $n = 2^k$, `fft1` and `fft2` both do $(k-1)\frac{n}{2}$ multiplications. Duplicating of powers of ω means all recursive calls in `fft1` and `fft2` access W sequentially.

Codes like these where the recursive calls update separate parts of the array A are easy to parallelize using Cilk. To parallelize `fft1` in Cilk C we first make a new subroutine `fft1cilk` which executes the two recursive calls in parallel. In Figure 2 the two Cilk `spawn` directives do this. For small n we do not want to start two new processes because of the cost of process management overhead. Thus for $n \leqslant 2^{15}$, `fft1cilk` calls `fft1` which runs sequentially.

To obtain good parallel speedup we also need to parallelize the for loop in `fft1` because it does $\frac{n}{2}$ multiplications out of a total of $(k-1)\frac{n}{2}$ multiplications. We do this by executing large contiguous blocks of the loop in parallel. We could use Cilk's parallel for loop construct `cilk_for` and let Cilk determine what block size to use but this does not allow us to tune the block size for best performance. Instead we choose a blocksize `B` and parallelize the loop explicitly as shown in Figure 2. To get best performance, we reduced `B` until the Cilk process overhead takes more than 1% say of the total time for the FFT. This yields good parallel speedup for many cores when n is large.

An alternative way to parallelize an FFT is to use a recursive FFT for creating parallel tasks for large n as we have but use a non-recursive FFT for smaller n. These and other strategies for parallelizing the FFT for multi-core computers are discussed in [9, 13].

4.2 Step 4: the Taylor shift

Steps 1, 2, 3, 5, and 6 of Algorithm 6 require $O(d)$ time. Steps 3 and 6 involve $d-1$ inverses in $\mathbb{F}_p$ which, because they require many divisions, are relatively very expensive. All the inverses can be replaced with multiplications as

```
void fft1( LONG *A, LONG n,
           LONG *W, LONG p ) {
   LONG i,n2,s,t;
   if( n==1 ) return;
   if( n==2 ) {
      s = addmod(A[0],A[1],p);
      t = submod(A[0],A[1],p);
      A[0] = s; A[1] = t;
      return;
   }
   n2 = n/2;
   fft1( A,    n2, W+n2, p );
   fft1( A+n2, n2, W+n2, p );
   for( i=0; i < n2; i++ ) {
      s = A[i];
      t = mulmod(W[i],A[n2+i],p);
      A[   i] = addmod(s,t,p);
      A[n2+i] = submod(s,t,p);
   }
   return;
}
```

```
void fft2( LONG *A, LONG n,
           LONG *W, LONG p ) {
   LONG i,n2,s,t;
   if( n==1 ) return;
   if( n==2 ) {
      s = addmod(A[0],A[1],p);
      t = submod(A[0],A[1],p);
      A[0] = s; A[1] = t;
      return;
   }
   n2 = n/2;
   for( i=0; i < n2; i++ ) {
      s = addmod(A[i],A[n2+i],p);
      t = submod(A[i],A[n2+i],p);
      A[   i] = s;
      A[n2+i] = mulmod(t,W[i],p);
   }
   fft2( A,    n2, W+n2, p );
   fft2( A+n2, n2, W+n2, p );
   return;
}
```

Figure 1: C code for two FFTs over $\mathbb{F}_p$.

follows. After computing $d!$ in step 1, we compute the inverse $\texttt{finv} = (d!)^{-1}$. For steps 3 and 6, we run the loop backwards and multiply by i to get the next inverse as follows.

```
for( i=d; i>1; i-- ) { Pi[i] = mulmod(Pi[i],finv,p); finv = mulmod(finv,i,p); }
```

We parallelized only the polynomial multiplication $\tilde{L}E$ in step 4. To multiply two polynomials of degree at most d our FFT multiplication needs three arrays A, B, and W of size n where $n = 2^k$ satisfies $2d < n \leqslant 4d$.

The data in Table 4 shows that the best parallel speedup for step 3 is 4.12. A factor of 4.12 on 10 cores is not very good and due to the fact that we did not paralellize steps 1, 3 and 6 which involve $2d$, $3d$ and $2d$ multiplications in $\mathbb{F}_p$, respectively. We did not do this because computing $P(z+\tau)$ is not a bottleneck of Algorithm 1.

Of course, it is possible to paralellize steps 1, 3 and 6. For q blocks of size B we would need to compute the factorials $F = [B!, (2B)!, (3B)!, \ldots, (qB)!]$ in parallel. We could do this by first computing q partial products $[\,B!, \prod_{k=B+1}^{2B} k, \prod_{k=2B+1}^{3B} k, \ldots, \prod_{k=qB+1}^{d} k\,]$ in parallel and then compute F. Now we can execute step 1 in q blocks in parallel. Next we would compute the inverses of F so we can execute steps 3 and 6 in parallel.

4.3 Step 5: the tangent Graeffe loop

Step 5 of Algorithm 1 is the main computation in Algorithm 1. It has complexity $NO(\mathsf{M}(d)) = O(\mathsf{M}(d)\log(p/s))$. We successfully parallelized the FFTs and all $O(d)$ loops in blocks except the following data movement:

```
for( i=0; i<n/2; i++ ) A[i] = A[2*i];   // compress
for( i=0; i<n/2; i++ ) A[n/2+i] = A[i]; // duplicate
```

We tried parallelizing the above code by moving the even entries of A into a temporary array T, in blocks, in parallel, then copying them back to A, twice, in blocks, in parallel. On 6 cores, the parallel code is slower than the serial code so we have left this pure data movement not parallelized.

4.4 Step 6: parallelizing the evaluations

Each of the three evaluations $A(\omega^i)$, $A'(\omega^i)$ and $B(\omega^i)$ for $0 \leqslant i < s$ in step 6 of Algorithm 1 can be done using the Bluestein transform [4] in $O(\mathsf{M}(s)) = O(\mathsf{M}(d))$ time. Although step 6 does not dominate the time complexity

```
cilk void block1( LONG n2, LONG *A, LONG b, LONG W, LONG p ) {
   LONG i,s,t;
   for( i=0; i<b; i++ ) {
      s = A[i];
      t = mulmod(W[i],A[n2+i],p);
      A[   i] = addmod(s,t,p);
      A[n2+i] = submod(s,t,p);
   }
   return;
}
#define B 65536
#define FFTCUTOFF 32768
cilk void fft1cilk( LONG *A, LONG n, LONG *W, LONG p ) {
   LONG q,r,i;
   if( n<=FFTCUTOFF ) { fft1(A,n,W,p); return; }
   spawn fft1cilk( A,    n2, W+n2, p );
   spawn fft1cilk( A+n2, n2, W+n2, p );
   sync; // wait for the two fft1cilk calls to finish
   n2 = n/2; q = n2/B; r = n2-q*B;
   for( i=0; i<q; i++ ) spawn block1( n2, A+i*B, B, W+i*B, p );
   if( r>0 ) spawn block1( n2, A+q*B, r, W+q*B, p );
   sync; // wait for all blocks to complete
   return;
}
```

Figure 2: Cilk C code for parallelizing `fft1`

it needs the most space. We need at least $3s$ units of storage to store the values of $A(\omega^i)$, $A'(\omega^i)$ and $B(\omega^i)$. For $p = 5 \times 2^{55} + 1$ and $d = 10^9$, the requirement $4d \leqslant s < 8d$ implies $s = 5 \times 2^{30}$. This is 120 gigabytes. We chose to save half of this by choosing $2d \leqslant s < 4d$ instead. This increases the time a little (see Table 3) because we obtain a smaller portion of the roots at each recursive call of Algorithm 1.

The Bluestein transform does one multiplication of two polynomials of degree s plus $O(s)$ work. For a multiplication of degree s we need an FFT of size $n = 2^k > 2s$ and our FFT uses $3n$ units of temporary storage. For $p = 5 \times 2^{55} + 1$, $d = 10^9$ and $s = 5 \times 2^{29}$ we have $n = 2^{33}$ so we need 192 gigabytes. We do not evaluate of $A(z)$, $A'(z)$ and $B(z)$ in parallel as that would triple the 192 gigabytes to 576 gigabytes.

For $p = r2^k + 1$ with r small, because $s = r2^j$ for some j, we can save time and space by applying (6). We do 2^j DFTs of size r, in blocks, in parallel, then s multiplications by powers of ω, in blocks in parallel, followed by r FFTs of size s/r which we do in parallel. This reduces the time for an evaluation by a factor between 6 and 12 since we replace 3 FFTs of size n where $2s < n \leqslant 4s$ by r FFTs of size s/r which is equivalent to one FFT of size s.

We need an additional s units of storage for storing the r inputs of size 2^j for the r FFTs and an additional s/r units for the required W array. Thus applying (6) reduces the temporary storage needed from $3n$ units to $s + s/r$ units. For $p = 5 \times 2^{55} + 1$ and $d = 10^9$, with $s = 5 \times 2^{29}$, this is $8(s + s/r) = 24$ gigabytes of temporary storage instead of 192 gigabytes.

We can now state the total storage needed by our implementation of Algorithm 1. We need to store the input $P(z)$ of degree d and an array of size d for the output roots S. We need space for three temporary polynomials of degree d. Including the memory for step 6, our implementation uses $5d+4+3s+s+s/r$ units, which, for $p = 52^{55}+1$, $d = 10^9$ and $s = 5 \times 2^{29}$ is 121 gigabytes.

4.5 Step 9: parallelizing the product tree multiplication algorithm

In step 9 we need to expand the polynomial $Q = \prod_{\alpha \in S}(z - \alpha)$. Let $n = |S|$ and suppose S is stored in an array of size n and Q in an array of size $n + 1$. We can compute Q in $O(\mathsf{M}(n) \log n)$ using fast multiplication and divide and conquer. The obvious way to do this is to split the roots in S into two equal size groups, of size $m = \lfloor n/2 \rfloor$ and

$d = n - m$, then recursively compute

$$a(x) = \prod_{i=1}^{m}(z - S_i) = x^m + \sum_{i=0}^{m-1} a_i z^i \text{ and } b(x) = \prod_{i=m+1}^{n}(z - S_i) = x^d + \sum_{i=0}^{d-1} b_i z^i$$

and finally multiply $a(x)$ by $b(x)$ using fast multiplication for large n. In the literature this approach is called the product tree multiplication algorithm; see [17, Algorithm 10.3]. In Appendix A, the function `mult` implements the product tree algorithm in Magma. We use it to create the input polynomial $P(z)$ to get Magma timings. However, `mult` is inefficient because all recursive calls allocate new memory for the inputs and products in the product tree.

For large products we use our FFT based polynomial multiplication `FFTpolmul64s` which needs a temporary array T of size $3N$ units of storage where $N = 2^k > n$. We do not want to allocate space for T in each recursive call. Fortunately, N is a power of 2 so we can divide T into two halves and execute the two recursive calls in parallel using the two halves of T. But the size of $a(x)$ plus $b(x)$ is $n + 2$. They don't fit in Q. The remedy is to note that a and b are both monic, so we do not need to store their leading coefficients.

Figure 3 presents C code for an in-place product tree algorithm which uses the input space S of size n, the output space Q of size $n + 1$ and a temporary array T of size $3N$ for all polynomial multiplications in the product tree. Our solution splits S asymmetrically into a "large" set of size $m = 2^k$ where $\frac{n}{2} \leqslant m < n$ and a "small" set of size $d = n - m$. For example, for $n = 36$ we use $m = 32$ and $d = 4$. To multiply $a \times b$ we first recursively compute $\hat{a} := a - x^m$ in $Q[0..m-1]$ and $\hat{b} := b - x^d$ in $Q[m..n-1]$. We copy the contents of Q to the input array S and we compute the product $\hat{a} \times \hat{b}$ in $Q[0..n-2]$. For small n, we use `polmul64s` an in-place multiplication with quadratic complexity. Then we add $x^d\hat{a}$ and $x^m\hat{b}$ to Q so that Q contains $\widehat{ab} := ab - x^n$. Thus we have the following result.

Proposition 6 *Let $S \subset \mathbb{F}_p$ of size n and Q be an array of size $n+1$ and T an array of size $3N$ where $N = 2^k$ and $N > n$. Assuming $N|(p-1)$, Algorithm `treemul` computes $Q = \prod_{\alpha \in S} z - \alpha$ in $O(\mathsf{M}(n) \log n)$ arithmetic operations in $\mathbb{F}_p$ using only the memory of S, Q and T.*

In Figure 3, because the memory for S, Q and T in the two recursive calls is separated, we may execute the two recursive calls to `treemul` in parallel. This also holds recursively, so we can keep executing recursive calls in parallel, which is ideal for Cilk. In our parallel implementation of `treemul` we parallelize the two recursive calls if $n \geqslant 2^{16} = 65536$. We also parallelize `FFTmul64s` and the FFTs (both the subalgorithms and their calls inside `treemul`). We do not parallelize the two polynomial additions.

Remark 10 By dividing the input S asymmetrically so that the FFTs are fully utilized, we gain upto a factor of 2 in speed when d is much smaller then m. Thus we get the benefit of using a truncated FFT [22] for the polynomial multiplications without using it.

Remark 11 If we insert the statement `Q[n] = 1;` before the `return` statement in `treemul`, then the sequential code will still work. However, since both recursive calls update `Q[n]`, the memory is no longer separated, so the two recursive calls cannot be executed in parallel.

5 Timing Results

We want to compare our new tangent Graeffe implementation with an implementation of the Cantor–Zassenhaus algorithm [8] which uses fast polynomial arithmetic. One of the best implementations that we know of is Magma's `Factorize` command. Magma uses an implementation of Shoup's work [44] by Allan Steel [45]. For $P(z) \in \mathbb{F}_p[z]$ of degree d with d distinct roots in $\mathbb{F}_p$, Magma uses the Cantor–Zassenhaus algorithm which has time complexity $O(\mathsf{M}(d) \log d \log p)$.

The timings in Tables 1 and 2 were obtained on an 8 core Intel Xeon E5-2660 CPU which runs at at 2.2 GHz base and 3.0 GHz turbo. The input polynomials $P(z)$ in Tables 1 and 2 of degree d were created with d random distinct roots in $\mathbb{F}_p$. Table 1 is for the 28.8 bit prime $p = 7 \times 2^{26} + 1$ and Table 2 is for the 62.4 bit prime $p = 3 \times 29 \times 2^{56} + 1$.

The timings in column NewTG are for our C implementation of Algorithm 1 with the parameter s chosen in $[2d, 4d)$. The timings in column Magma are for the `Factorize` command for Magma version V2.25-5. Magma code for creating $P(z)$ and factoring it is given in Appendix 6. We note that older versions of Magma have a quadratic sub-algorithm; this problem was fixed by Steel for version V2.25-5. We also note that Magma has faster arithmetic for primes $p < 2^{30}$.

```
#include <stdlib.h>
#define LONG long long int

// Input polynomials a and b mod p of degree da and db respectively.
// Output array c is space for the sum a + b mod p (product a b mod p)
LONG poladd64s( LONG *a, LONG *b, LONG *c, LONG da, LONG db, LONG p );
LONG polmul64s( LONG *a, LONG *b, LONG *c, LONG da, LONG db, LONG p );
LONG FFTmul64s( LONG *a, LONG *b, LONG *c, LONG da, LONG db, LONG *T, LONG p );

void treemul( LONG *S, LONG n, LONG *Q, LONG *T, LONG p ) {
  LONG i,m,d;
  if( n==1 ) { Q[0] = submodp(0,S[0],p); return; }
  for( m=1; 2*m<n; m=2*m );
  d = n-m;
  treemul(S,m,Q,T,p);  // compute a(x)-x^m in Q[0..m-1] using S[0..m-1] and T[0..3m-1]
  treemul(S+m,d,Q+m,T+3*m,p);  // and b(x)-x^d in Q[m..n-1] using S[m..n-1] and T[3m..]
  for( i=0; i<n; i++ ) S[i] = Q[i];   // S = [a0,a1,...,am-1,b0,b1,...,bd-1]
  if( d<10 ) polmul64s(S,S+m,Q,m-1,d-1,p); // in-place classical multiplication
  else FFTmul64s(S,S+m,Q,m-1,d-1,T,p);  // FFT multiplication using T[0..6m-1]
  Q[n-1] = 0;
  poladd64s(Q+d,S,Q+d,m-1,m-1,p);    // add x^m (b-x^d) to Q[d..n-1]
  poladd64s(Q+m,S+m,Q+m,d-1,d-1,p);  // add x^d (a-x^m) to Q[m..n-1]
  // Q[n] = 1; is okay for this sequential code but not for a parallel code
  return;
}

LONG *array64s(LONG n); // return an array of size n
void treeproduct( LONG *S, LONG n, LONG *Q, LONG p ) {
  // S is an array of size n and Q of size n+1
  LONG N,*T;
  for( N=1; N<n; N=2*N );
  T = array64s(3*N);
  treemul(S,n,Q,T,p);
  free(T);
  Q[n] = 1;
  return;
}
```

Figure 3: C code for computing $Q = \prod_{i=0}^{n-1}(z - S_i)$ in step 9.

The columns in Tables 1 and 2 labeled Step5, Step6, Step9, and P/Q report the time spent in steps 5, 6, 9, and 10, respectively, in the top level call of Algorithm 1. They show that initially step 5, which costs $O(\mathsf{M}(d)\log(p/s)$, dominates the cost of steps 6 and 9 which cost $O(\mathsf{M}(s))$ and $O(\mathsf{M}(d)\log d)$ respectively. As d increases, so does s, and the number of iterations N of step 5 in Algorithm 1 drops; ultimately, the computation of the product Q in step 9 dominates.

The column labeled %roots reports the percentage of the roots found in the first tangent Graeffe application. Column N is the value of N in step 5. Step 5 does $4N + 2$ FFTs of size $n = d + 1$. Column FFT reports the time for one of those FFTs. For example, in Table 2, for $d = 2^{20} - 1$, 13.9 seconds was spent in step 5. The code did $4 \times 41 + 2 = 166$ FFTs of size $n = 2^{20}$ which took $166 \times 0.0747 = 12.4$ seconds.

The timings in Table 1 show that the tangent Graeffe method is 100 times faster than Magma's implementation of Cantor–Zassenhaus at degree $2^{20} - 1$. For the larger prime in Table 2 the tangent Graeffe method is 166 times faster at degree $2^{20} - 1$.

Table 3 shows the effect of different choices for the parameter s in Algorithm 1. If the roots of $P(z + \tau)$ under the Graeffe transform G_{2^N} are uniformly distributed among the s-th roots of unity, then the proportion of them that remain distinct is $\mathrm{e}^{-d/s}$. Thus doubling s increases the proportion of roots found at each recursive level of Algorithm 1, which saves time, but it also doubles the cost of the evaluations in step 6. Recall that the theoretically

d	Magma	NewTG	Step5	Step6	Step9	P/Q	first	%roots	N	FFT
$2^{13}-1$	3.38 s	0.07 s	0.02 s	0.01 s	0.01 s	0.00 s	0.05 s	75.0%	14	0.41 ms
$2^{14}-1$	7.34 s	0.17 s	0.05 s	0.03 s	0.01 s	0.01 s	0.11 s	74.4%	13	0.86 ms
$2^{15}-1$	16.96 s	0.32 s	0.10 s	0.05 s	0.03 s	0.02 s	0.23 s	75.1%	12	1.84 ms
$2^{16}-1$	39.05 s	0.64 s	0.20 s	0.10 s	0.08 s	0.03 s	0.46 s	75.7%	11	3.85 ms
$2^{17}-1$	88.65 s	1.21 s	0.37 s	0.19 s	0.17 s	0.06 s	0.92 s	75.3%	10	7.93 ms
$2^{18}-1$	203.0 s	2.54 s	0.71 s	0.43 s	0.41 s	0.12 s	1.94 s	75.2%	9	16.8 ms
$2^{19}-1$	462.5 s	5.16 s	1.34 s	0.90 s	0.91 s	0.25 s	3.94 s	75.1%	8	35.2 ms
$2^{20}-1$	1050. s	10.5 s	2.51 s	1.84 s	2.03 s	0.52 s	8.01 s	75.2%	7	74.7 ms
$2^{21}-1$	2407. s	21.2 s	4.55 s	3.80 s	4.41 s	1.11 s	16.1 s	75.4%	6	157.5 ms
$2^{22}-1$	—	43.0 s	8.14 s	8.06 s	9.62 s	2.31 s	32.7 s	75.7%	5	333.7 ms
$2^{23}-1$	—	86.2 s	13.7 s	16.3 s	21.1 s	4.91 s	64.4 s	76.3%	4	697.5 ms
$2^{24}-1$	—	174.0 s	22.4 s	33.5 s	46.3 s	10.2 s	132. s	77.5%	3	1460. ms
$2^{25}-1$	—	342.8 s	32.2 s	67.2 s	99.7 s	20.7 s	258. s	80.1%	2	3055. ms

Table 1: Timings in CPU seconds for $p = 7 \times 2^{26} + 1$ using $s \in [2d, 4d)$.

d	Magma	NewTG	Step5	Step6	Step9	P/Q	first	%roots	N	FFT
$2^{13}-1$	18.94 s	0.21 s	0.08 s	0.04 s	0.01 s	0.00 s	0.14 s	69.8%	48	0.41 ms
$2^{14}-1$	44.07 s	0.46 s	0.18 s	0.08 s	0.01 s	0.01 s	0.30 s	68.8%	47	0.86 ms
$2^{15}-1$	103.5 s	0.98 s	0.37 s	0.17 s	0.03 s	0.02 s	0.64 s	69.2%	46	1.83 ms
$2^{16}-1$	234.2 s	2.06 s	0.77 s	0.35 s	0.08 s	0.05 s	1.33 s	68.9%	45	3.83 ms
$2^{17}-1$	534.5 s	4.15 s	1.54 s	0.72 s	0.17 s	0.08 s	2.67 s	69.2%	44	7.93 ms
$2^{18}-1$	1219. s	8.90 s	3.24 s	1.53 s	0.38 s	0.18 s	5.70 s	69.2%	43	16.6 ms
$2^{19}-1$	2809. s	18.69 s	6.75 s	3.24 s	0.87 s	0.40 s	12.0 s	69.2%	42	35.2 ms
$2^{20}-1$	6428. s	38.76 s	13.9 s	6.79 s	1.93 s	0.85 s	24.9 s	69.2%	41	74.7 ms
$2^{21}-1$	—	79.88 s	28.3 s	14.2 s	4.11 s	1.77 s	51.4 s	69.2%	40	157.5 ms
$2^{22}-1$	—	165.5 s	57.6 s	29.8 s	9.01 s	3.71 s	106. s	69.2%	39	333.5 ms
$2^{23}-1$	—	335.4 s	115. s	60.8 s	19.2 s	7.66 s	215. s	69.2%	38	697.5 ms
$2^{24}-1$	—	702.5 s	238. s	129. s	42.4 s	16.3 s	451. s	69.2%	37	1460. ms

Table 2: Timings in CPU seconds for $p = 3 \times 29 \times 2^{56} + 1$ using $s \in [2d, 4d)$.

optimal performance is obtained by taking $s/d \asymp \sqrt{\log d}$, where we note that $\sqrt{\log d} = 3.730$ and $\sqrt{\log_2 d} = 4.480$ for $d = 1{,}100{,}000$.

Column $e^{-d/s}$ gives the expected proportion of roots that will be found. Column %roots is the actual percentage of roots found by Algorithm 1. Columns Step5, Step6, and Total are the time in step 5, step 6, and the total time. The data in Table 3 suggests choosing $4d \leqslant s < 8d$ to minimize time. For $4d \leqslant s < 8d$ the algorithm is expected to obtain between $e^{-1/4} = 0.779$ and $e^{-1/8} = 0.882$ of the roots. Our actual choice of $2d \leqslant s < 4d$ increases the time by about 10% but saves a factor of 2 in space.

	$d = 1{,}100{,}000$					$d = 1{,}400{,}000$				
Choice of s	$e^{-d/s}$	%roots	Step5	Step6	Total	$e^{-d/s}$	%roots	Step5	Step6	Total
$d \leqslant s < 2d$	0.432	43.3%	27.6 s	0.54 s	64.53 s	0.586	58.5%	27.4 s	0.96 s	67.10 s
$2d \leqslant s < 4d$	0.657	65.7%	27.3 s	1.16 s	44.71 s	0.766	76.5%	26.7 s	2.39 s	47.17 s
$4d \leqslant s < 8d$	0.811	81.1%	26.7 s	2.40 s	39.86 s	0.875	87.4%	25.9 s	4.92 s	43.14 s
$8d \leqslant s < 16d$	0.900	90.0%	25.9 s	4.92 s	39.47 s	0.935	93.5%	24.0 s	10.2 s	45.34 s
$16d \leqslant s < 32d$	0.949	94.9%	25.2 s	10.3 s	43.16 s	0.967	96.7%	24.4 s	21.2 s	55.12 s

Table 3: Tangent Graeffe timings in CPU seconds for $p = 5 \times 2^{55} + 1$ for various s.

Table 4 shows the results for our parallel implementation including timings for $P(z)$ of degree 10^9. The timings in Table 4 were obtained on a server with a 10 core Intel E5 2680 v2 CPU with 128 gigabytes of RAM running at 3.0GHz base, 3.6GHz turbo. The input polynomial $P(z)$ is created as before by choosing d random distinct values

from $\mathbb{F}_p$ with $p = 5 \times 2^{55} + 1$.

d	cores	$P(z)$	Step3	Step5	Step6	Step8	Step9	P/Q	first	Total	Space
10^6	1	1.931	0.393	8.446	0.841	0.624	1.233	0.562	12.11	19.44	120 MiB
10^6	10	0.292	0.135	1.320	0.147	0.078	0.190	0.157	2.035	5.627	120 MiB
speedup		6.61x	2.91x	6.40x	5.72x	8.00x	6.49x	3.58x	5.95x	3.45x	
$2 \cdot 10^6$	1	3.975	0.817	17.18	1.671	1.246	2.683	1.180	24.79	39.97	240 MiB
$2 \cdot 10^6$	10	0.538	0.261	2.421	0.297	0.150	0.397	0.273	3.823	8.336	240 MiB
speedup		7.39x	3.13x	7.10x	5.63s	8.31x	6.76x	4.32x	6.48x	4.79x	
$4 \cdot 10^6$	1	8.458	1.713	35.00	3.533	2.512	5.826	2.475	51.09	82.31	488 MiB
$4 \cdot 10^6$	10	1.110	0.533	5.043	0.625	0.300	0.797	0.521	7.845	16.25	488 MiB
speedup		7.62x	3.21x	6.94x	5.65x	8.37x	7.31x	4.75x	6.51x	5.07x	
$8 \cdot 10^6$	1	18.16	3.533	70.75	7.244	5.001	12.63	5.220	104.4	168.6	0.95 GiB
$8 \cdot 10^6$	10	2.383	1.070	10.07	1.218	0.599	1.680	1.045	15.74	27.66	0.95 GiB
speedup		7.62x	3.30x	7.03x	5.95x	8.35x	7.52x	5.00x	6.63x	6.10x	
$25 \cdot 10^7$	1	819.4	133.3	2316	274.2	157.2	574.0	203.4	3660	5815	30.3 GiB
$25 \cdot 10^7$	10	102.7	37.49	319.7	45.75	18.54	74.29	38.50	536.1	850.4	30.3 GiB
speedup		7.98x	3.56x	7.23x	5.99x	8.48x	7.73x	5.28x	6.83x	6.84x	
$5 \cdot 10^8$	1	1752	274.9	4631	566.6	316.7	1228	422.5	7442	11820	60.6 GiB
$5 \cdot 10^8$	10	219.7	75.66	636.0	92.46	37.35	159.7	76.92	1082	1719.3	60.6 GiB
speedup		7.79x	3.63x	7.28x	6.12x	8.48x	7.69x	5.49x	6.88x	6.88x	
10^9	1	3739	645.2	10486	1423	682.6	2795.5	1017	17057	26889	121 GiB
10^9	10	465.2	156.5	1366	243.8	83.03	341.5s	180.7	2379	3714.7	121 GiB
speedup		8.04x	4.12x	7.67x	5.88x	8.22x	8.19x	5.63x	7.17x	7.23x	

Table 4: Real timings (in seconds) for $p = 5 \times 2^{55} + 1$ using $s \in [2d, 4d)$.

Timings in column $P(z)$ are for computing the input polynomial $P(z)$ using our in-place parallel product tree multiplication from section 3.4. The total time for computing the roots of $P(z)$ is in column Total. Roughly speaking, our software takes 10 times longer to compute the roots of $P(z)$ (less for larger d) than it does to create $P(z)$!

Timings in columns Step4, Step5, Step6, Step9, and P/Q are the times for those steps for the top-level call of Algorithm 1. Speedups are given for each computation. The reader can see that the parallel speedup is not as good for Step4 and the division P/Q. This is because we have only parallelized the FFTs in them; none of steps of linear cost are parallelized.

6 Conclusion

The motivation for our work is the problem of sparse polynomial interpolation where one seeks to recover the coefficients and monomials of a polynomial $f \in \mathbb{F}_p[x_1, \ldots, x_n]$ with t terms from values of f. The popular approach by Prony and Ben-Or–Tiwari needs only $2t$ values of f but it needs to factor a polynomial $\Lambda(z) \in \mathbb{F}_p[t]$ of degree t which has t distinct roots in $\mathbb{F}_p$. Using the Cantor–Zassenhaus (CZ) algorithm, computing the roots of $\Lambda(z)$ takes $O(\mathsf{M}(t) \log_2 t \log_2 p)$ time. Because this is the most expensive step in Ben-Or/Tiwari sparse interpolation, and because CZ does gcd computations which are difficult to parallelize, research in sparse interpolation has tried different approaches that do not require root finding.

If we choose the prime p of the form $p = \sigma 2^k + 1$ with σ small, the new tangent Graeffe (TG) algorithm factors $\Lambda(z)$ in $O(\mathsf{M}(t) \log_2 p)$ time. This is $\Theta(\log t)$ faster than CZ, but the constants actually matter in practice. In this work we improved the main step of TG by a factor of 2 and we showed that for large p, TG is faster than CZ by a factor of $7/2 \log_2 t$. Our new C implementation of TG is over 100 times faster than Magma's C implementation of CZ for $t > 2^{20}$ on the tests we made. So TG really is a lot faster than CZ.

Another goal was to see if we could parallelize TG for multi-core computers. We found that it was not sufficient to only parallelize the underlying FFT. We also had to parallelize many sub-algorithms to get good parallel speedup. Here we contributed a new parallel in-place product tree algorithm. We were also successful in reducing the space needed so that we could compute the roots of $\Lambda(z)$ of degree one billion on a 10 core computer with 128 gigabytes of RAM in about one hour. It should be possible to reduce the time further by using vectorization [26, 12]. The

sequential implementation of the FFTs could also be further improved using techniques from [15, 42] and Harvey's approach for number theoretic FFTs [21].

Appendix A Magma code

```
p := 3*29*2^56+1;
p := 7*2^26+1;
Fp := FiniteField(p);
R<x> := PolynomialRing(Fp);

mult := function( L )
  n := #L;
  if n eq 1 then return x-L[1];
  else
     m := n div 2;
     f := $$( L[1..m] ) * $$( L[m+1..n] );
     return f;
  end if;
end function;

d := 2^12-1;
S := { Random(Fp) : i in [1..d] };
while #S lt d do S := S join { Random(Fp) : i in [1..d-#S] }; end while;
#S;
L := [ x : x in S ];
time f := mult( L );
time g := Factorization(f);
```

References

[1] A. V. Aho, K. Steiglitz, and J. D. Ullman. Evaluating polynomials on a fixed set of points. *SIAM Journ. of Comp.*, 4:533–539, 1975.

[2] M. Ben-Or and P. Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 301–309. New York, NY, USA, 1988. ACM Press.

[3] D.J. Bernstein. *Fast multiplication and its applications*, pages 325–384. Mathematical Sciences Research Institute Publications. Cambridge University Press, United Kingdom, 2008.

[4] Leo I. Bluestein. A linear filtering approach to the computation of discrete Fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 18(4):451–455, 1970.

[5] A. Bostan, G. Lecerf, and É. Schost. Tellegen's principle into practice. In *ISSAC '03: Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*, pages 37–44. ACM Press, 2003.

[6] R. P. Brent, F. G. Gustavson, and D. Y. Y. Yun. Fast solution of Toeplitz systems of equations and computation of Padé approximants. *J. Algorithms*, 1(3):259–295, 1980.

[7] J. Canny, E. Kaltofen, and Y. Lakshman. Solving systems of non-linear polynomial equations faster. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation*, pages 121–128. ACM Press, 1989.

[8] D. G. Cantor and H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Math. Comp.*, 36(154):587–592, 1981.

[9] E. Chu and A. George. *Inside the FFT Black Box*. Computational Mathematics Series. CRC Press, 2000.

[10] S. Czapor, K. Geddes, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[11] A. Díaz and E. Kaltofen. FOXFOX: a system for manipulating symbolic objects in black box representation. In *ISSAC '98: Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation*, pages 30–37. ACM Press, 1998.

[12] P. Fortin, A. Fleury, F. Lemaire, and M. Monagan. High performance SIMD modular arithmetic for polynomial evaluation. *ArXiv:2004.11571*, 2020.

[13] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. Moura. Discrete Fourier transform on multicores. *IEEE Signal Processing Magazine*, 26(6):90–102, 2009.

[14] T. S. Freeman, G. M. Imirzian, E. Kaltofen, and Y. Lakshman. DAGWOOD: a system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Software*, 14:218–240, 1988.

[15] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[16] M. Frigo, C.E. Leisorson, and R.K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of PLDI 1998*, pages 212–223. ACM, 1998.

[17] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 3rd edition, 2013.

[18] P. Giorgi, B. Grenet, and D. S. Roche. Fast in-place algorithms for polynomial operations: division, evaluation, interpolation. Technical Report `http://arxiv.org/abs/2002.10304`, Arxiv, 2020.

[19] B. Grenet, J. van der Hoeven, and G. Lecerf. Randomized root finding over finite fields using tangent Graeffe transforms. In *Proc. ISSAC '15*, pages 197–204. New York, NY, USA, 2015. ACM.

[20] B. Grenet, J. van der Hoeven, and G. Lecerf. Deterministic root finding over finite fields using Graeffe transforms. *AAECC*, 27(3):237–257, 2016.

[21] D. Harvey. Faster arithmetic for number-theoretic transforms. *J. Symbolic Comput.*, 60:113–119, 2014.

[22] J. van der Hoeven. The truncated Fourier transform and applications. In J. Gutierrez, editor, *Proc. ISSAC 2004*, pages 290–296. Univ. of Cantabria, Santander, Spain, July 4–7 2004.

[23] J. van der Hoeven. Probably faster multiplication of sparse polynomials. Technical Report, HAL, 2020. `http://hal.archives-ouvertes.fr/hal-02473830`.

[24] J. van der Hoeven and G. Lecerf. On the bit-complexity of sparse polynomial and series multiplication. *J. Symbolic Comput.*, 50:227–254, 2013.

[25] J. van der Hoeven and G. Lecerf. Sparse polynomial interpolation in practice. *ACM Commun. Comput. Algebra*, 48(3/4):187–191, 2015.

[26] J. van der Hoeven, G. Lecerf, and G. Quintin. Modular SIMD arithmetic in Mathemagix. *ACM Trans. Math. Softw.*, 43(1):5–1, 2016.

[27] J. van der Hoeven and M. Monagan. Implementing the tangent Graeffe root finding method. In A. M. Bigatti, J. Carette, J. H. Davenport, M. Joswig, and T. de Wolff, editors, *Mathematical Software – ICMS 2020*, pages 482–492. Cham, 2020. Springer International Publishing.

[28] J. van der Hoeven et al. GNU TeXmacs. `https://www.texmacs.org`, 1998.

[29] J. Hu and M. B. Monagan. A fast parallel sparse polynomial GCD algorithm. In S. A. Abramov, E. V. Zima, and X.-S. Gao, editors, *Proc. ISSAC '16*, pages 271–278. ACM, 2016.

[30] M. A. Huang and A. J. Rao. Interpolation of sparse multivariate polynomials over large finite fields with applications. In *SODA '96: Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 508–517. Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[31] M. Javadi and M. Monagan. Parallel sparse polynomial interpolation over finite fields. In *Proceedings of PASCO 2010*, pages 160–168. ACM Press, 2010.

[32] E. Kaltofen. Computing with polynomials given by straight-line programs I: greatest common divisors. In *STOC '85: Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, pages 131–142. ACM Press, 1985.

[33] E. Kaltofen, Y. N. Lakshman, and J.-M. Wiley. Modular rational sparse multivariate polynomial interpolation. In *ISSAC '90: Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 135–139. New York, NY, USA, 1990. ACM Press.

[34] E. Kaltofen and B. M. Trager. Computing with polynomials given by black boxes for their evaluations: greatest common divisors, factorization, separation of numerators and denominators. *JSC*, 9(3):301–320, 1990.

[35] E. Kaltofen and L. Yagati. Improved sparse multivariate polynomial interpolation algorithms. In *ISSAC '88: Proceedings of the International Symposium on Symbolic and Algebraic Computation*, pages 467–474. Springer Verlag, 1988.

[36] R. Larrieu. The truncated Fourier transform for mixed radices. In *Proc. ISSAC '17*, pages 261–268. New York, NY, USA, 2017. ACM.

[37] R. Moenck. Fast computation of GCDs. In *Proc. of the 5th ACM Annual Symposium on Theory of Computing*, pages 142–171. New York, 1973. ACM Press.

[38] N. Möller and T. Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60:165–175, 2011.

[39] H. Murao and T. Fujise. Modular algorithm for sparse multivariate polynomial interpolation and its parallel implementation. *JSC*, 21:377–396, 1996.

[40] J. M. Pollard. The fast Fourier transform in a finite field. *Mathematics of Computation*, 25(114):365–374, 1971.

[41] R. Prony. Essai expérimental et analytique sur les lois de la dilatabilité des fluides élastiques et sur celles de la force expansive de la vapeur de l'eau et de la vapeur de l'alkool, à différentes températures. *J. de l'École Polytechnique Floréal et Plairial, an III*, 1(cahier 22):24–76, 1795.

[42] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[43] D. S. Roche. What can (and can't) we do with sparse polynomials? In *Proc. ISSAC '18*, pages 25–30. New York, NY, USA, 2018. ACM.

[44] V. Shoup. A new polynomial factorization and its implementation. *J. Symbolic Computation*, 20(4):363–397, 1995.

[45] A. Steel. Private communication.