Parallel Sparse Polynomial Interpolation over Finite Fields

Seyed Mohammad Mahdi Javadi

School of Computing Science, Simon Fraser University, Burnaby, B.C. Canada

Michael Monagan

Department of Mathematics, Simon Fraser University, Burnaby, B.C. Canada

Abstract

We present a probabilistic algorithm to interpolate a sparse multivariate polynomial over a finite field, represented with a black box. Our algorithm modifies the algorithm of Ben-Or and Tiwari from 1988 for interpolating polynomials over rings with characteristic zero to characteristic p by doing additional probes.

To interpolate a polynomial in n variables with t non-zero terms, Zippel's (1990) algorithm interpolates one variable at a time using $O(ndt)$ probes to the black box where d bounds the degree of the polynomial. Our new algorithm does $O(nt)$ probes. It interpolates each variable independently using $O(t)$ probes which allows us to parallelize the main loop giving an advantage over Zippel's algorithm.

We have implemented both Zippel's algorithm and the new algorithm in C and we have done a parallel implementation of our algorithm using Cilk [3]. In the paper we provide benchmarks comparing the number of probes our algorithm does with both Zippel's algorithm and Kaltofen and Lee's hybrid of the Zippel and Ben-Or/Tiwari algorithms.

Key words: sparse polynomial interpolation, parallel interpolation algorithms, Ben-Or Tiwari.

⋆ This work was supported by NSERC of Canada and the MITACS NCE of Canada

Email addresses: sjavadi@cecm.sfu.ca (Seyed Mohammad Mahdi Javadi), mmonagan@cecm.sfu.ca (Michael Monagan).

URLs: http://www.sfu.ca/~sjavadi (Seyed Mohammad Mahdi Javadi), http://www.cecm.sfu.ca/~monaganm/ (Michael Monagan).

1. Introduction

Let p be a prime and $f \in \mathbb{Z}_p[x_1, \ldots, x_n]$ be a multivariate polynomial with $t > 0$ non-zero terms which is represented by a *black box* $\mathbf{B} : \mathbb{Z}_p^n \to \mathbb{Z}_p$. On input of $(\alpha_1, \ldots, \alpha_n) \in \mathbb{Z}_p^n$, the black box evaluates and outputs $f(x_1 = \alpha_1, \ldots, x_n = \alpha_n)$. Given also a degree bound $d \geq \deg f$ on the degree of f, our goal is to interpolate the polynomial f with minimum number of evaluations (probes to the black box). Newton interpolation needs $O(n^{d+1})$ points to interpolate f which is exponential in d. For sparse f, that is, $t \ll n^{d+1}$, we seek algorithms whose computational complexity is polynomial in t, n, d and $\log p$.

Sparse interpolation plays a key role in several algorithms in computer algebra such as algorithms for polynomial GCD computation in $\mathbb{Z}[x_1, x_2, \ldots, x_n]$ and solving systems of polynomial equations involving parameters over $\mathbb{Q}$. In these applications one solves the problems modulo a prime p where p is usually chosen to be a machine prime, typically 31 or 63 bits. In 1979 Richard Zippel in [30] presented the first sparse interpolation algorithm which he developed to solve the GCD problem. It makes $O(ndt)$ probes to the black box. Zippel's algorithm is probabilistic. It relies heavily on the assumption that if a polynomial is zero at a *random* evaluation point, then it is the zero polynomial with high probability. In 1990, Zippel in [31] improved his algorithm by using evaluation points of the form $(\alpha_1^i, \ldots, \alpha_k^i) \in \mathbb{Z}_p^k$ so that the linear systems to be solved become transposed Vandermonde systems which can be solved in $O(t^2)$ time and $O(t)$ space instead of $O(t^3)$ time and $O(t^2)$ space – see [17]. Zippel's algorithm is used as the main algorithm for GCD computation in $\mathbb{Z}[x_1, x_2, \ldots, x_n]$ in several computer algebra systems, including Mathematica, Maple (see [24]), and Magma. For multivariate GCD computation over algebraic function fields over $\mathbb{Q}$, in [15], we used Zippel's interpolation algorithm in to interpolate variables and parameters.

In 1994 Rayes, Wang and Weber in [28] looked at parallelizing Zippel's algorithm. However, because it interpolates f one variable at a time, sequentially, it's parallelism is limited. This was our motivation for looking for a new approach that we present in this paper. Our approach is based on the sparse interpolation of Ben-Or and Tiwari which we now describe.

In 1988, Ben-Or and Tiwari [2] presented a deterministic algorithm for interpolating a multivariate polynomial with integer, rational, real or complex coefficients. Given a bound T on the number of terms t of the polynomial f, the algorithm evaluates the black box at powers of the first n primes; it evaluates at the points $(2^i, 3^i, 5^i, \ldots, p_n^i)$ for $0 \leq i < 2T$. If $M_j(x_1, \ldots, x_n)$ are the monomials of the t non-zero terms of f, it then uses Berlekamp/Massey algorithm [23] from coding theory and a root finding algorithm to find the monomial evaluations $M_j(2, 3, 5, \ldots, p_n)$ for $1 \leq j \leq t$ and then determines the degree of each monomial M_j in x_k by trial division of $M_j(2, 3, 5, \ldots, p_n)$ by p_k. The major disadvantage of the Ben-Or/Tiwari algorithm for $\mathbb{Z}[x_1, \ldots, x_n]$ is that the evaluation points are large ($O(T \log n)$ bits long − see [2]). Moreover, a severe expression swell occurs when the Berlekamp-Massey algorithm is run over $\mathbb{Q}$ which makes the algorithm very slow. This was addressed by Kaltofen *et al.* in [18] by running the algorithm modulo a power of a prime of sufficiently large size; the modulus must be greater than $\max_j M_j(2, 3, 5, \ldots, p_n)$.

The Ben-Or/Tiwari algorithm will work in finite field of characteristic p without modification when $p > p_n^d$. For small finite fields, Grigoriev, Karpinski, and Singer in [10] propose a parallel algorithm which uses field extensions of degree $2 \log_q[nt] + 3$. This

method is not practical for large q as it requires inversion of a general q by q matrix. In [14], Huang and Rao describe how to make the Ben-Or/Tiwari approach work over finite fields GF(q) with at least $4t(t-2)d^2+1$ elements. Their idea is to replace the primes $2,3,5,\ldots,p_n$ in Ben-Or/Tiwari by linear (hence irreducible) polynomials in GF(q)[y]. Their algorithm is Las Vegas and does $O(dt^2)$ probes. Although the authors discuss how to parallelize the algorithm, the factor of t^2 means it's not practical for large t.

In 2000, Kaltofen et al. in [19, 20] presented a hybrid of the Zippel and Ben-Or/Tiwari algorithms, which they call a "racing algorithm". To reduce the number of probes when interpolating the next variable in Zippel's algorithm, their algorithm runs Newton interpolation and univariate Ben-Or/Tiwari simultaneously, stopping when the first succeeds. However, this further sequentializes the algorithm. In Section 5, we compare the number of probes made by this algorithm to our new algorithm. Experimentally, for random sparse polynomials, we find that it makes approximately $2nt$ probes.

In 2009, Giesbrecht, Labahn and Lee in [9] presented two new algorithms for sparse interpolation for polynomials with floating point coefficients. The first is a modification of the Ben-Or/Tiwari algorithm that uses $O(t)$ probes. To avoid numerical problems, it evaluates at powers of complex roots of unity of relatively prime order. In principle, this algorithm can be made to work over finite fields $GF(p)$ for applications where one can choose the prime p. One needs $p-1$ to have n distinct prime factors $p_1,p_2,\ldots,p_n$ all $> d$. Given a primitive element α and elements $\omega_1,\omega_2,\ldots,\omega_n$ of order $p_1,p_2,\ldots,p_n$ in $GF(p)$, the exponents $(e_1,e_2,\ldots,e_n)$ of the value of a monomial $m=\omega_1^{e_1}\omega_2^{e_2}\ldots\omega_n^{e_n}$ can be obtained from the discrete logarithm; $e_i=\beta \mod p_i$ where $\beta=\log_\alpha(m)$. Finding such primes is not difficult. For example, for $n=6$, $d=30$, we find $31,33,35,37,41$ are the first five relatively prime integers greater than d. Let $q=54316185$ be their product. We find $r=58$ is the first even integer satisfying $r>d$, $\gcd(r,q)=1$, and $p=rq+1$ is prime. Now for such primes discrete logarithms in $GF(p)$ can be done efficiently using the Pohlig-Hellman algorithm [26]. The prime p has 31.6 bits in length. In general, the prime $p>(d+1)^n$ thus the length of the prime depends linearly on the number of variables. We have not explored the feasibility of this approach.

In 2010 Kaltofen [21] suggested a similar approach. He first reduces multivariate interpolation to univariate interpolation using the Kronecker substitution $(x_1,x_2,\ldots,x_n)=(x,x^{d+1},\ldots,x^{(d+1)^n})$ and interpolates the univariate polynomial from powers of a primitive element α using a prime $p>(d+1)^n$. If p is smooth, that is, $p-1$ has no large prime factors, then the discrete logarithms are efficient and the complexity is polynomial in the logarithm of the degree. This approach has the added advantage that by choosing p smooth of the form $p=2^k s+1$, one can directly use the FFT in GF(p) when needed elsewhere in the algorithm.

In 2009 Garg and Schost [5] presented an interpolation algorithm to interpolate over any commutative ring S with identity which is polynomial in $\log d$. For sparse univariate $g(x)$ in $S[x]$ they evaluate $g(x)$ modulo $x^{p_i}-1$ for N primes $p_1,p_2,\ldots,p_N$, also exploiting the roots of unity. The result that they obtain requires $N>T^6\log d$ probes, thus too many probes to be practical for large t. For multivariate $f(x_1,x_2,\ldots,x_n)$ they also use the Kronecker substitution. A similar idea is used in [11] by Giesbrecht and Roche for interpolating shifted-lacunary polynomials.

Our approach for sparse interpolation over $\mathbb{Z}_p$ is to use evaluation points of the form $(\alpha_1^i,\ldots,\alpha_n^i)\in\mathbb{Z}_p^n$ and modify the Ben-Or/Tiwari algorithm to do extra probes to determine the degrees of the variables in each monomial in f. We do $O(nt)$ probes in order

to recover the monomials from their images. The main advantage of our approach is the increased parallelism.

Our paper is organized as follows. In Section 2 we present an example showing the main flow and the key features of our algorithm. Our algorithm is probabilistic. We identify possible problems that can occur and how the new algorithm deals with them in Section 3. In Section 4 we present our new algorithm and analyze its sequential time complexity and failure probability. We then present two optimizations and an improvement for root finding over finite fields. Finally, in Section 5 we compare the C implementations of our algorithm and Zippel's algorithm with the racing algorithm of Kaltofen and Lee [20] on various sets of polynomials. A preliminary version of this work appeared in [16]. In comparison with our work in [16], we have redesigned the algorithm in this paper to improve it's parallel performance.

2. The Idea and an Example

Let $f = \sum_{i=1}^{t} C_i M_i \in \mathbb{Z}_p[x_1, \ldots, x_n]$ be the polynomial represented with the black box $\mathbf{B} : \mathbb{Z}_p^n \rightarrow \mathbb{Z}_p$ with $C_i \in \mathbb{Z}_p \backslash \{0\}$. Here t is the number of non-zero terms in f. $M_i = x_1^{e_{i1}} \times x_2^{e_{i2}} \times \cdots \times x_n^{e_{in}}$ is the i'th monomial in f where $M_i \neq M_j$ for $i \neq j$. Let $T \geq t$ be a bound on the number of non-zero terms and let $d \geq \deg f$ be a bound on the degree of f so that $d \geq \sum_{j=1}^{n} e_{ij}$ for all $1 \leq i \leq t$. We demonstrate our algorithm on the following example. Here we use x, y and z for variables instead of x_1, x_2 and x_3.

Example 1. Let $f = 91yz^2 + 94x^2yz + 61x^2y^2z + 42z^5 + 1$ and $p = 101$. Here $t = 5$ terms and $n = 3$ variables. We suppose we are given a black box that computes f and we want to interpolate f. We will use $T = 5$ and $d = 5$ for the term and degree bounds. The first step is to pick n non-zero elements $\alpha_1, \alpha_2, \ldots, \alpha_n$ from $\mathbb{Z}_p$ at random. We evaluate the black box at the points

$$(\alpha_1^i, \alpha_2^i, \ldots, \alpha_n^i) \text{ for } 0 \leq i < 2T.$$

Thus we make $2T$ probes to the black box. Let $V = (v_0, v_1, \ldots, v_{2T-1})$ be the output. For our example, for random evaluation points $\alpha_1 = 45, \alpha_2 = 6$ and $\alpha_3 = 69$ we obtain $V = (87, 26, 15, 94, 63, 15, 49, 74, 43, 71)$.

Now we use the Berlekamp/Massey algorithm [23] (See [19] for a more accessible reference). The input to this algorithm is a sequence of elements $s_0, s_1, \ldots, s_{2t-1}, \ldots$ from any field F. The algorithm computes a *linear generator* for the sequence, i.e. the univariate polynomial $\Lambda(z) = z^t - \lambda_{t-1}z^{t-1} - \cdots - \lambda_0$ such that

$$s_{t+i} = \lambda_{t-1}s_{t+i-1} + \lambda_{t-2}s_{t+i-2} + \cdots + \lambda_0 s_i \text{ for all } i \geq 0.$$

In our example where $F = \mathbb{Z}_p$, the input is $V = (v_0, \ldots, v_{2T-1})$ and the output is

$$\Lambda_1(z) = z^5 + 80\,z^4 + 84\,z^3 + 16\,z^2 + 74\,z + 48.$$

In the next step, we choose n non-zero $(b_1, \ldots, b_n) \in \mathbb{Z}_p^n$ at random such that $b_k \neq \alpha_k$ for all $1 \leq k \leq n$. In this example we choose $b_1 = 44, b_2 = 9, b_3 = 18$. Now we choose the evaluation points $(b_1^i, \alpha_2^i, \ldots, \alpha_n^i)$ for $0 \leq i < 2T - 1$. Note that this time we are evaluating the first variable at powers of b_1 instead of α_1. We evaluate the black box at these points and apply the Berlekamp/Massey algorithm on the sequence of the outputs to compute the second linear generator

$$\Lambda_2 = z^5 + 48\,z^4 + 92\,z^3 + 9\,z^2 + 91\,z + 62.$$

We repeat the above process for a new set of evaluation points $(\alpha_1^i, b_2^i, \alpha_3^i, \ldots, \alpha_n^i) \in \mathbb{Z}_p^n$, i.e., we replace α_2 by b_2 obtaining

$$\Lambda_3 = z^5 + 42\,z^4 + 73\,z^3 + 73\,z^2 + 73\,z + 41$$

the third linear generator. Similarly, for evaluation points $(\alpha_1^i, \alpha_2^i, b_3^i)$ we compute

$$\Lambda_4 = z^5 + 73\,z^4 + 8\,z^3 + 94\,z^2 + 68\,z + 59.$$

Note that we can compute $\Lambda_1, \ldots, \Lambda_{n+1}$ in parallel. We know (see [2]) that if the monomial evaluations are distinct over $\mathbb{Z}_p$ for each set of evaluation points, then $\deg_z(\Lambda_i) = t$ for all $1 \leq i \leq n$ and each Λ_i has t non-zero roots in $\mathbb{Z}_p$. Ben-Or and Tiwari prove that for each $1 \leq i \leq t$, there exists $1 \leq j \leq t$ such that

$$m_i = M_i(\alpha_1, \ldots, \alpha_n) \equiv r_{0j} \bmod p.$$

where $r_{01}, \ldots, r_{0t}$ are the roots of Λ_1. In the next step we compute $r_{(i-1)1}, \ldots, r_{(i-1)t}$ the roots of the Λ_i. We have

$$\begin{aligned}
&\{r_{01} = 1, r_{02} = 50, r_{03} = 84, r_{04} = 91, r_{05} = 98\} \text{ (roots of } \Lambda_1) \\
&\{r_{11} = 1, r_{12} = 10, r_{13} = 69, r_{14} = 84, r_{15} = 91\} \text{ (roots of } \Lambda_2) \\
&\{r_{21} = 1, r_{22} = 25, r_{23} = 69, r_{24} = 75, r_{25} = 91\} \text{ (roots of } \Lambda_3) \\
&\{r_{31} = 1, r_{32} = 8, r_{33} = 25, r_{34} = 35, r_{35} = 60\} \text{ (roots of } \Lambda_4)
\end{aligned}$$

The main step now is to determine the degrees of each monomial M_i of f in each variable. Consider the first variable x. We know that $m_i' = M_i(b_1, \alpha_2, \ldots, \alpha_n)$ is a root of Λ_2 for $1 \leq i \leq n$. On the other hand we have

$$\frac{m_i'}{m_i} = \frac{M_i(b_1, \alpha_2, \ldots, \alpha_n)}{M_i(\alpha_1, \alpha_2, \ldots, \alpha_n)} = \left(\frac{b_1}{\alpha_1}\right)^{e_{i1}}. \tag{1}$$

Let $r_{0j} = M_i(\alpha_1, \alpha_2, \ldots, \alpha_n)$ and $r_{1k} = M_i(b_1, \alpha_2, \ldots, \alpha_n)$. From Equation 1 we have

$$r_{1k} = r_{0j} \times \left(\frac{b_1}{\alpha_1}\right)^{e_{i1}},$$

i.e. for every root r_{0j} of Λ_1, $r_{0j} \times \left(\frac{b_1}{\alpha_1}\right)^{e_{i1}}$ is a root of Λ_2 for some e_{i1} which is the degree of some monomial in f with respect to x. This gives us a way to compute the degree of each monomial M_i in the variable x.

In this example we have $\frac{b_1}{\alpha_1} = 93$. We start with the first root of Λ_1 and check if $r_{01} \times \left(\frac{b_1}{\alpha_1}\right)^i$ is a root of Λ_2 for $0 \leq i \leq d$. To do this one could simply evaluate the polynomial $\Lambda_2(z)$ at $z = r_{01}(\frac{b_1}{\alpha_1})^i$ and see if we get 0. This costs $O(t)$ operations. Instead we compute and sort the roots of Λ_2 so we can do this using binary search in $O(\log t)$. For $r_{01} = 1$ we have $r_{01} \times \left(\frac{b_1}{\alpha_1}\right)^0 = 1$ is a root of Λ_2, and, for $0 < i \leq d$, $r_{01} \times \left(\frac{b_2}{\alpha_1}\right)^i$ is *not* a root of Λ_2, hence we conclude that the degree of the first monomial of f in x is 0. We continue this to find the degrees of all the monomials in f in the variable x. We obtain

$$e_{11} = 0, e_{21} = 2, e_{31} = 0, e_{41} = 0, e_{51} = 2.$$

We proceed to the next variable y. Again using the same approach as above, we find that the degrees of the monomials in the second variable y to be

$$e_{12} = 0, e_{22} = 1, e_{32} = 1, e_{42} = 0, e_{52} = 2.$$

Similarly we compute the degrees of other monomials in z:

$$e_{13} = 0, e_{23} = 1, e_{33} = 2, e_{43} = 5, e_{53} = 1.$$

At this point we have computed all the monomials. Recall that $M_i = x_1^{e_{i1}} \times x_2^{e_{i2}} \times \cdots \times x_n^{e_{in}}$ hence we have

$$M_1 = 1, M_2 = x^2yz, M_3 = yz^2, M_4 = z^5, M_5 = x^2y^2z.$$

The reader may observe that once $\Lambda_1(z)$ is computed, determining the degrees of the monomials M_i in each variable represent n independent tasks which can be done in parallel. This is a key advantage of our algorithm.

Now we need to compute the coefficients. We do this by solving one linear system. We computed the roots of Λ_1 and we have computed the monomials such that $M_i(\alpha_1, \ldots, \alpha_n) = r_{0i}$. Recall that v_i is the output of the black box on input $(\alpha_1^i, \ldots, \alpha_n^i)$ hence we have

$$v_i = C_1 r_{01}^i + C_2 r_{02}^i + \cdots + C_t r_{0t}^i$$

for $0 \leq i \leq 2t-1$. This linear system is a Vandermonde system which can be solved in $O(t^2)$ time and $O(t)$ space (see [31]). After solving we obtain

$$C_1 = 1, C_2 = 94, C_3 = 91, C_4 = 42 \text{ and } C_5 = 61$$

and hence $g = 1 + 94x^2yz + 91yz^2 + 42z^5 + 61x^2y^2z$ is our interpolated polynomial. We will show later that for p sufficiently large, $g = f$ with high probability. However, we can also check whether $g = f$ with high probability as follows; we choose evaluation points $(\alpha_1, \ldots, \alpha_n)$ at random and test if $\mathbf{B}(\alpha_1, \ldots, \alpha_n) = g(\alpha_1, \ldots, \alpha_n)$. If the results match, the algorithm returns g as the interpolated polynomial, otherwise it fails.

3. Problems

The evaluation points $\alpha_1, \ldots, \alpha_n$ must satisfy certain conditions for our new algorithm to output f. Here we identify all problems.

3.1. Distinct Monomials

The first condition is that for $1 \leq i \neq j \leq t$

$$M_i(\alpha_1, \ldots, \alpha_n) \neq M_j(\alpha_1, \ldots, \alpha_n) \text{ in } \mathbb{Z}_p$$

so that $\deg(\Lambda_1(z)) = t$. Also, at the k'th step of the algorithm, when computing the degrees of the monomials in x_k, we must have for all $1 \leq i \neq j \leq t$

$$m_{i,k} \neq m_{j,k} \text{ in } \mathbb{Z}_p \text{ where } m_{i,k} = M_i(\alpha_1, \ldots, \alpha_{k-1}, b_k, \alpha_{k+1}, \ldots, \alpha_n)$$

so that $\deg(\Lambda_{k+1}(z)) = t$. We now give an upper bound on the probability that no monomial evaluations collide when we use random non-zero elements of $\mathbb{Z}_p$ for evaluations.

Lemma 1. Let $\alpha_1, \ldots, \alpha_n$ be random non-zero evaluation points in $\mathbb{Z}_p$ and let $m_i = M_i(\alpha_1, \ldots, \alpha_n)$. Then the probability that two different monomials evaluate to the same value (we get a collision) is

$$\text{Prob}(m_i = m_j : 1 \leq i < j \leq t) \leq \binom{t}{2} \frac{d}{(p-1)} < \frac{dt^2}{2(p-1)}.$$

Proof. Consider the polynomial

$$A = \prod_{1 \le i < j \le t} \left(M_i(x_1, \ldots, x_n) - M_j(x_1, \ldots, x_n) \right).$$

Observe that $A(\alpha_1, \ldots, \alpha_n) = 0$ iff two monomial evaluations collide. Recall that the Schwartz-Zippel lemma ([29, 30]) says that if $r_1, \ldots, r_n$ are chosen at random from any subset S of a field K and $F \in K[x_1, \ldots, x_n]$ is non-zero then

$$\text{Prob}(F(r_1, \ldots, r_n) = 0) \le \frac{\deg F}{|S|}.$$

Our result follows from noting that $d \ge \deg f$ and thus $\deg A \le \binom{t}{2} d$ and $|S| = p - 1$ since we choose α_i to be non-zero from $\mathbb{Z}_p$. □

Remark 1. If any of the $\alpha_i = 1$ then the probability of monomial evaluations colliding is clearly high. To reduce the probability of monomial evaluations colliding, in an earlier version of our algorithm, we picked α_i to have order $> d$. We did this by picking random generators of $\mathbb{Z}_p^*$. There are $\phi(p-1)$ generators where ϕ is Euler's totient function. However, if one does this, the restriction on the choice of α leads to a weaker result, namely, $\binom{t}{2} \frac{d}{\phi(p-1)}$.

3.2. Root Clashing

Let $r_{01}, \ldots, r_{0t}$ be the roots of $\Lambda_1(z)$ which is the output of the Berlekamp/Massey algorithm using the first set of evaluation points $(\alpha_1^i, \ldots, \alpha_n^i)$ for $0 \le i < 2T$. To compute the degrees of all the monomials in the variable x_k, as mentioned in the Example 1, the first step is to compute Λ_{k+1}. Then if $\deg_{x_k}(M_i) = e_{ik}$ we have $r_{ki} = r_{0i} \times (\frac{b_k}{\alpha_k})^{e_{ik}}$ is a root of Λ_{k+1}. If $r_{0i} \times (\frac{b_k}{\alpha_k})^{e'}$, $0 \le e' \ne e_{ik} \le d$ is also a root of Λ_{k+1} then we have a root clash and we cannot uniquely identify the degree of the monomial M_i in x_k.

Example 2. Consider the polynomial given in Example 1. Suppose instead of choosing $b_1 = 44$, we choose $b_1 = 72$. Since α_1, α_2 and α_3 are the same as before, Λ_1 does not change and hence the roots of Λ_1 are $r_{01} = 1, r_{02} = 7, r_{03} = 41, r_{04} = 61$ and $r_{05} = 64$. In the next step we substitute $b_1 = 72$ for α_1 and compute $\Lambda_2 = z^5 + 61z^4 + 39z^3 + 67z^2 + 37z + 98$. We proceed to compute the degrees of the monomials in x but we find that

$$r_4 \times (\frac{\alpha_4}{\alpha_1})^2 = 15 \quad \text{and} \quad r_4 \times (\frac{\alpha_4}{\alpha_1})^4 = 7$$

are both roots of Λ_2, hence we can not determine the degree of the last monomial in x.

Lemma 2. If $\deg \Lambda_1(z) = \deg \Lambda_{k+1}(z) = t$ then the probability that there is a root clash, that is, we can not uniquely compute the degrees of all the monomials $M_i(x_1, \ldots, x_n)$ in x_k is at most $\frac{d(d+1)t^2}{4(p-2)}$.

Proof. Let $S_i = \{r_{0j} \times (\frac{b_k}{\alpha_k})^i \mid 1 \le j \le t\}$ for $0 \le i \le d$. We assume that $r_{0i} \ne r_{0j}$ for all $1 \le i \ne j \le t$. We will not be able to uniquely identify the degree of the j'th monomial in x_k if there exists $\bar{d}$ such that $r_{0j} \times (\frac{b_k}{\alpha_k})^{\bar{d}} = r_{ki}$ is a root of $\Lambda_{k+1}(z)$ and $0 \le d \ne e_{jk} \le d$ where e_{jk} is $\deg_{x_k}(M_j)$. But we have $r_{ki} = r_{0i} \times (\frac{b_k}{\alpha_k})^{e_{ik}}$ thus $r_{0j} \times (\frac{b_k}{\alpha_k})^{\bar{d}} = r_{0i} \times (\frac{b_k}{\alpha_k})^{e_{ik}}$. Without loss of generality, assume $\tilde{d} = \bar{d} - e_{ik} > 0$. We have $r_{0i} = r_{0j} \times (\frac{b_k}{\alpha_k})^{\tilde{d}}$ and hence

$r_{0i} \in S_{\tilde{d}} \Rightarrow S_0 \cap S_{\tilde{d}} \neq \emptyset$. Hence we will not be able to compute the degrees in x_k if $S_0 \cap S_i \neq \emptyset$ for some $1 \leq i \leq d$. Let

$$g(x) = \prod_{1 \leq l \neq j \leq t} (r_{0j}x^i - r_{0l}\alpha_k^i).$$

We have $r_{0l} = r_{0j} \times (\frac{b_k}{\alpha_k})^i \in S_0 \cap S_i$ iff $g(b_k) = 0$. Applying the Schwartz-Zippel lemma, the probability that $g(b_k) = 0$ is at most $\frac{\deg g}{|S|} = \frac{\binom{t}{2} i}{(p-2)} < \frac{it^2}{2(p-2)}$ since we chose $b_k \neq \alpha_k \neq 0$ at random from $\mathbb{Z}_p$. If we sum this quantity for all $1 \leq i \leq d$ we obtain that the overall probability is at most $\frac{d(d+1)t^2}{4(p-2)}$. $\square$

4. The Algorithm

Algorithm: Parallel Interpolation

Input: A *black box* $\mathbf{B} : \mathbb{Z}_p^n \to \mathbb{Z}_p$ that on input $\alpha_1, \ldots, \alpha_n \in \mathbb{Z}_p^n$ outputs $f(\alpha_1, \ldots, \alpha_n)$ where $f \in \mathbb{Z}_p[x_1, \ldots, x_n] \backslash \{0\}$.

Input: A degree bound $d \geq \deg(f)$.

Input: A bound $T \geq t$ on the number of terms in f. (*For reasonable probability of success we require* $p > dT^2$.)

Output: The polynomial f or FAIL.

1: Choose $\alpha_1, \ldots, \alpha_n$ from $\mathbb{Z}_p \backslash \{0\}$ at random.
2: **for** k from 0 to n in **parallel do**
3: Case $k = 0$: Compute $\Lambda_1(z)$ using $(\alpha_1, \ldots, \alpha_n)$:
Evaluate the black box $\mathbf{B}$ at $(\alpha_1^i, \ldots, \alpha_n^i) \in \mathbb{Z}_p^n$ for $0 \leq i \leq 2T-1$ and apply the Berlekamp Massey algorithm to the sequence of $2T$ outputs.
4: Case $k > 0$: Choose non-zero $b_k \in \mathbb{Z}_p \backslash \{0\}$ at random until $b_k \neq \alpha_k$ and compute $\Lambda_{k+1}(z)$ using $\alpha_1, \ldots, a_{k-1}, b_k, \alpha_{k+1}, \ldots, \alpha_n$.
5: **end for**
6: Set $t = \deg \Lambda_1(z)$. If the degree of the Λ's are not all equal to t then **return** FAIL.
7: **for** k from 0 to n in **parallel do**
8: Compute $\{r_{k1}, \ldots, r_{kt}\}$ the set of distinct roots of $\Lambda_{k+1}(z)$.
9: **end for**
10: **for** k from 1 to n in **parallel do**
11: Determine $\deg_{x_k}(M_i)$ for $1 \leq i \leq t$ as described in Section 2. If we failed to to compute the degrees uniquely (see Section 3.2) then **return** FAIL.
12: **end for**
13: Let $S = \{C_1 r_{01}^i + C_2 r_{02}^i + \cdots + C_t r_{0t}^i = v_i \mid 0 \leq i \leq 2t-1\}$. Solve the linear system S for $(C_1, \ldots, C_t) \in \mathbb{Z}_p^t$ and set $g = \sum_{i=1}^t C_i M_i$ where $M_i = \prod_{j=1}^n x_j^{e_{ij}}$.
14: Choose $\alpha_1, \ldots, \alpha_n$ from $\mathbb{Z}_p \backslash \{0\}$ at random.
If $\mathbf{B}(\alpha_1, \ldots, \alpha_n) \neq g(\alpha_1, \ldots, \alpha_n)$ then **return** FAIL.
15: **return** g.

Remark 2. The algorithm presented corresponds to our parallel implementation in Cilk. Further parallelism is available. In particular, one may compute the $2T$ probes to the black box $\mathbf{B}$ in step 3 in parallel. We remark that Kaltofen in [22] pointed out to us that assuming $T \geq t$, then $T + t$ probes are sufficient to determine any $\Lambda(z)$ and that the Berlekamp-Massey algorithm can be modified to stop after processing $T + t$ inputs.

Remark 3. The algorithm is probabilistic. If the degrees of the Λ's are all equal but less than t then monomial evaluations have collided and the algorithm cannot compute f. The check in step 14 detects incorrect g with probability at least $1 - d/(p-1)$ (the Schwartz-Zippel lemma). Thus by doing one additional probe to the black box, we verify the output g with high probability. Kaltofen and Lee in [20] also use additional probes to verify the output this way.

Theorem 1. If $(p-2) > 2^{k-2}3(n+1)d(d+3)t^2$ then Algorithm Parallel Interpolation outputs $f(x_1, \ldots, x_n)$ with probability at least $1 - 1/2^k$. Moreover, the probability that the algorithm outputs an incorrect result is less than $\frac{d}{p-1} \times \left(\frac{dt^2}{2(p-1)}\right)^n$.

Proof. Algorithm Parallel Interpolate will need $(n+1)$ Λ's all of degree t. The choice of α_k, and b_k must be non-zero and distinct. Thus applying Lemmas 1 and 2, the probability that all $n+1$ Λ's have degree t and we can compute all the monomial degrees with no collisions is at least $1 - \frac{(n+1)d\,t^2}{2(p-2)} - \frac{n\,d(d+1)t^2}{4(p-2)} > 1 - \frac{3(n+1)d(d+3)t^2}{4(p-2)}$. Solving $1 - \frac{3(n+1)d(d+3)t^2}{4(p-2)} > 1 - 2^{-k}$ for $p-2$ gives the first result. For the second result, the algorithm outputs an incorrect result only if all $\Lambda_1, \ldots, \Lambda_{n+1}$ have degrees less than t and the check in step 14 fails. This happens with probability less than $(\frac{dt^2}{2(p-1)})^n$ (see Lemma 1) and less than $\frac{d}{p-1}$ (see Remark 3), respectively. □

4.1. Complexity Analysis

We now give the sequential complexity of the algorithm in terms of the number of arithmetic operations in $\mathbb{Z}_p$. We need to consider the cost of probing the black box. Let $E(n,t,d)$ be the cost of one probe to the black box. We make $2(n+1)T$ probes in the first loop and one in step 14. Hence the cost of probes to the black box is $O(nTE(n,t,d))$. The $n+1$ calls to the Berlekamp/Massey algorithm in the first loop (as presented in [19]) cost $O(T^2)$ each. The Vandermonde system of equations at Step 13 can be solved in $O(t^2)$ using the method given in [31]. Note that as mentioned in [31], when inverting a $t \times t$ Vandemonde matrix defined by $k_1, \ldots, k_t$, one of the most expensive parts is to compute the master polynomial $M(z) = \prod_{i=1}^{t}(z - k_i)$. However, in our algorithm we can use the fact that $M(z) = \prod_{i=1}^{t}(z - r_{0i}) = \Lambda_1(z)$.

To compute the roots of $\Lambda_{k+1}(z)$ at Step 8, we use Rabin's Las Vegas algorithm from [27]. The idea of Rabin's algorithm is to split $\Lambda(z)$ using the following gcd in $\mathbb{Z}_p[z]$

$$g(x) = \gcd((z+\beta)^{(p-1)/2} - 1, \Lambda(z))$$

for β chosen at random from $\mathbb{Z}_p$. For $\Lambda(z)$ with degree t, if classical algorithms for polynomial multiplication, division and gcd are used for $\mathbb{Z}_p[z]$, the cost is dominated by the first split which has expected cost $O(t^2 \log p)$ (see Ch. 8 of Geddes et. al. [8]) arithmetic operations in $\mathbb{Z}_p$.

To compute the degree of the monomials in the variable x_k in Step 11 of the algorithm, we sort the roots of $\Lambda_1(z)$ and $\Lambda_{k+1}(z)$. Then checking if $r_{0i} \times (\frac{b_k}{\alpha_k})^{\bar{d}}$ is a root of $\Lambda_{k+1}(z)$ can be done in $O(\log t)$ using binary search. Hence the the degrees can be computed in $O(t \log t + dt \log t)$.

Theorem 2. The expected number of arithmetic operations in $\mathbb{Z}_p$ that Algorithm Parallel Interpolation does is

$$O(n(t^2 \log p + dt \log t + T^2 + TE(n,t,d))$$

using classical (quadratic) algorithms for polynomial arithmetic. For $T \in O(t)$ this simplifies to $O(n(t^2 \log p + dt \log t + tE(n,t,d))$.

Apart from the cost of the probes, the most expensive component of the algorithm is the computation of the roots of the $\Lambda(z)$'s, each of which costs $O(t^2 \log p)$ using classical arithmetic. It is well known that this can be improved to $O(\log t(M(t) \log p + M(t) \log t))$ using fast multiplication (see Algorithm 14.15 of von zur Gathen and Gerhard [6]) where $M(t)$ is cost of multiplication of polynomials of degree t in $\mathbb{Z}_p[z]$. In our implementation we have implemented this asymptotically fast root finding algorithm because we found that the root finding was indeed an expensive component of the cost. We present an improvement in section 4.3.

Similarly, the generator polynomial $\Lambda_k(z)$ can also be computed using the Fast Euclidean Algorithm in $O(M(t) \log t)$. See [6] Ch. 11 for a description of the Fast Euclidean Algorithm and [6] Ch. 7 for a description of how to compute $\Lambda_k(z)$. Furthermore, once the support (the monomials) for a polynomial are known, the coefficients can be determined in $O(M(t) \log t)$ time using fast multiplication (see van der Hoven and Lecerf [13]). This leads to a complexity, softly linear in T, of $O(n(M(t) \log p \log t + dt \log t + M(T) \log T + TE(n,t,d)))$.

If we choose p to be a Fourier prime then multiplication in $\mathbb{Z}_p[t]$ $M(t) \in O(t \log t)$ using the FFT. Hence the expected sequential complexity of our algorithm is $O(n[t \log^2 t \log p + dt \log t + T \log^2 T + TE(n,t,d)])$ arithmetic operations in $\mathbb{Z}_p$.

4.1.1. Zippel's Algorithm

For comparison, we will briefly discuss the number of probes Zippel's 1990 interpolation algorithm does. Let t_i be the number of terms in the target polynomial f after evaluating variables $x_{i+1}, \ldots, x_n$. We have $t_0 = 1$ and $t_n = t$. The number of probes to the black box using Zippel's algorithm is

$$d + 1 + (t_1 d + t_2 d + \ldots t_{n-1} d) = 1 + d \sum_{i=0}^{n-1} t_i.$$

Since $t_i \leq t_n$ for all $1 \leq i \leq n-1$, the number of probes is in $O(ndt)$.

Example 3. Let $f = x^{20} + y^{20} + z^{20} + 1$ with $p = 1009, d = 20$ and $t = 4$. Our new algorithm will do 33 probes to the black box while Zippel's does about 121 probes.

We expect Zippel's algorithm to perform better than our algorithm for *dense* target polynomials.

Lemma 3. Let f be a dense polynomial of degree d in each variable so that the number of terms in f is $t = (d+1)^n$. Then the number of probes to the black box in Zippel's algorithm is exactly t. In comparison, our algorithm does $2(n+1)t + 1$ probes.

Proof. Here we have $t_i = (d+1)^i$ thus the number of probes is $1 + d \times \prod_{i=0}^{n-1} (d+1)^i = 1 + d \times \frac{(d+1)^n - 1}{d+1-1} = (d+1)^n = t$. □

4.2. Optimizations

4.2.1. Computing the degrees of the monomials in the last variable.

The first optimization is to compute the degree of each monomial $M_i = x_1^{e_{i1}} x_2^{e_{i2}} \dots x_n^{e_{in}}$ in the last variable x_n without doing any more probes to the black box. Suppose we have computed the degree of M_i in x_k for $1 \leq k < n$. We know that $M_i(\alpha_1, \dots, \alpha_n)$ is equal to r_{0i}, a root of Λ_1. Hence $r_{0i} = \alpha_1^{e_{i1}} \cdot \alpha_2^{e_{i2}} \cdot \dots \cdot \alpha_n^{e_{in}}$. Since we know the degrees e_{ij} for $1 \leq j < n$ we can determine e_{in} by division of $r_{0i} \cdot (\alpha_1^{e_{i1}} \dots \alpha_{n-1}^{e_{in-1}})^{-1}$ by α_n. This reduces the total number of probes from $2(2n+1)t$ to $2(2n-1)t$ and increases the probability of success from $> 1 - \frac{3(n+1)d(d+3)t^2}{4(p-2)}$ to $> 1 - \frac{3nd(d+3)t^2}{4(p-2)}$.

4.2.2. Bipartite perfect matching.

We now present an improvement that will allow our algorithm to determine the degree of the monomial M_i in x_k even when $r_{0i} \times \frac{b_k}{\alpha_k}^{e'}$ is also a root of $\Lambda_{k+1}(z)$ in most cases. Note that we assume the monomial evaluations are distinct, i.e. $\forall\ 1 \leq i \neq j \leq t,\ m_{i,k} \neq m_{j,k}$.

Suppose we have computed Λ_{k+1} and we want to compute the degrees of the monomials in x_k and let $R_1 = \{r_{01}, \dots, r_{0t}\}$ be the set of roots of Λ_1 and $R_k = \{r_{k1}, \dots, r_{kt}\}$ be the set of roots of Λ_{k+1}. Let

$$D_j = \{(i, r) \mid 0 \leq i \leq d,\ r = r_{0j} \times (\frac{b_k}{\alpha_1})^i \in R_k\}.$$

D_j contains the set of all possible degrees of the j'th monomial M_j in the k'th variable x_k. We know that $(e_{jk}, r_{kj}) \in D_j$ and hence $|D_j| \geq 1$. If $|D_j| = 1$ for all $1 \leq j \leq t$, then the degrees are unique and this step of the algorithm is complete. Let G_k be a balanced bipartite graph defined as follows. G_k has two independent sets of nodes U and V each of size t. Nodes in U and V represent elements in R_1 and R_k respectively, i.e. $u_i \in U$ and $v_j \in V$ are labeled with r_{0i} and r_{kj}. We connect $u_i \in U$ to $v_j \in V$ with an edge of weight (degree) d_{ij} if and only if $(d_{ij}, r_{kj}) \in D_i$. We illustrate with an example.

Example 4. Let f be the polynomial given in Example 1 and suppose for some evaluation points $\alpha_1, \dots, \alpha_3$ and b_1 we obtain the graph G_1 as shown in Figure 1. Notice that this graph has a unique perfect matching, i.e., the set of edges $\{(r_{0i}, r_{1i}) \mid 1 \leq i \leq 5\}$. Thus the degrees of the 5 monomials in x must be are 0, 0, 0, 2, and 2.

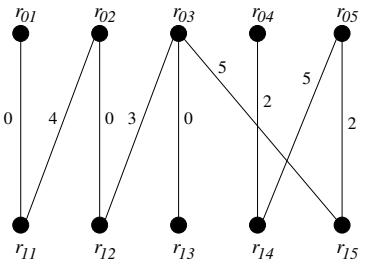


Fig. 1. The bipartite graph G_1

Lemma 4. We can uniquely identify the degrees of all the monomials in x_k if the bipartite graph G_k has a unique *perfect matching*.

Proof. Let $m_i = M_i(\alpha_1, \ldots, \alpha_n)$ and without loss of generality, assume $r_{0i} = m_i$ and $r_{ki} = m_{i,k}$ for all $1 \leq i \leq t$. We have $(e_{ik}, r_{ki}) \in D_i$ and hence $u_i \in U$ is connected to $v_j \in V$ in G_k with an edge of weight e_{ik}. This means that the set $S = \{(u_i, v_i, e_{ik}) \mid u_i \in U, v_i \in V\}$ is a perfect matching in G_k. If this perfect matching is unique then by finding it, we have computed e_{ik}'s, the degrees of the monomials in x_k. □

To find a prefect matching in the graph G_k one can use the Hopcroft–Karp algorithm [12]. This algorithm finds a matching in time $O(e\sqrt{v})$ where e and v are the number of edges and vertices respectively. However, for random sparse bipartite graphs, Bast *et al.* [1] (See also [25]) prove that the Hopcroft-Karp algorithm runs in time $O(e \log v)$ with high probability.

Lemma 5. If $p-1 > dT^2$ then the expected number of edges in the graph G_k is at most $(d+1)/4 + t$.

Proof. In lemma 2 we showed that the probability that there are no root clashes is greater than $1 - \frac{d(d+1)t^2}{4(p-2)}$. Therefore, the expected number of root clashes is less than $\frac{d(d+1)t^2}{4(p-2)}$ hence the expected number of edges of G_k is less than $t + \frac{d(d+1)t^2}{4(p-2)}$. If we choose p such that $p - 1 > dT^2$ then $p - 1 > dt^2$ and the expected number of edges in G_k is at less than $(d+1)/4 + t$. □

Thus if $4(d+1) \leq t$ then the expected number of edges is less than at most $2t$ hence G_k is sparse and the expected cost of finding a perfect matching would be $O(t \log t)$, which is softly linear in t.

4.2.3. Computing the degrees of the monomials in x_k.

Let $D = \deg(f)$. If the prime p is large enough, i.e. $p > \frac{nD(D+1)t^2}{4\epsilon}$ then with probability $1 - \epsilon$ the degree of every monomial in x_k can correctly be computed using only G_k and without needing any extra probes to the black box. In fact in this case, with high probability, every r_{0i} will be matched with exactly only one r_{kj} and hence every node in G_k would have degree one. But if $d \gg D$, i.e. the degree bound d is not tight, the probability that we could identify the degrees uniquely drops significantly even though p is large enough. This is because the probability that *root clashing* (see Section 3) happens, linearly depends on d. In this case, with probability $1 - \epsilon$, the degree of M_i in x_k would be $\min\{d_{ij} \mid (d_{ij}, r_i) \in G_k\}$, i.e. the edge connected to r_{0i} in G_k with minimum weight (degree) is our desired edge in the graph which will show up in the perfect matching. We apply Theorem 3.

Lemma 6. Let G_k be the bipartite graph for the k'th variable. Let $u_{i_1} \to v_{j_1} \to u_{i_2} \to v_{j_2} \to \cdots \to v_{j_s} \to u_{i_1}$ be a cycle in G_k where $u_l \in U$ is labeled with r_{0l} (a root of Λ_1) and $v_m \in V$ is labeled with r_{km} (a root of Λ_{k+1}). Let d_{lm} be the weight (degree) of the edge between u_l and v_m. We have $\sum_{m=1}^{s} d_{i_m j_m} - \sum_{m=1}^{s} d_{i_{m+1} j_m} = 0$.

Proof. It is easy to show that $r_{0i_1} = (\frac{b_k}{\alpha_k})^{\bar{d}} r_{0i_s}$ where $\bar{d} = d_{i_1 j_1} - d_{i_2 j_1} + d_{i_2 j_2} - d_{i_3 j_2} + \cdots + d_{i_{s-1} j_{s-1}} - d_{i_s j_{s-1}}$. Also both u_{i_1} and u_{i_s} are connected to v_{j_s} in G_k hence we have $r_{0i_1} = (\frac{b_k}{\alpha_k})^{d_{i_1 j_s}} r_{ki_s}$ and $r_{0i_s} = (\frac{b_k}{\alpha_k})^{d_{i_s j_s}} r_{ki_s}$. These three equations yield to $r_{0i_1} = (\frac{b_k}{\alpha_k})^{\tilde{d}} r_{0i_1}$ where $\tilde{d} = d_{i_1 j_1} - d_{i_2 j_1} + d_{i_2 j_2} - d_{i_3 j_2} + \cdots + d_{i_{s-1} j_{s-1}} - d_{i_s j_{s-1}} + d_{i_s j_s} - d_{i_1 j_s}$. But if $\frac{b_k}{\alpha_k}$

is of sufficiently high order, $\tilde{d}$ must be zero thus $\sum_{m=1}^{s} d_{i_m j_m} - \sum_{m=1}^{s} d_{i_{m+1} j_m} = 0$. $\square$

Example 5. In G_1' shown in Figure 2, there is a cycle $r_3 \to \tilde{r}_4 \to r_7 \to \tilde{r}_7 \to r_3$. The weights (degrees) of the edges in this cycle are as $7, 3, 0$ and 4. We have $7-3+0-4=0$.

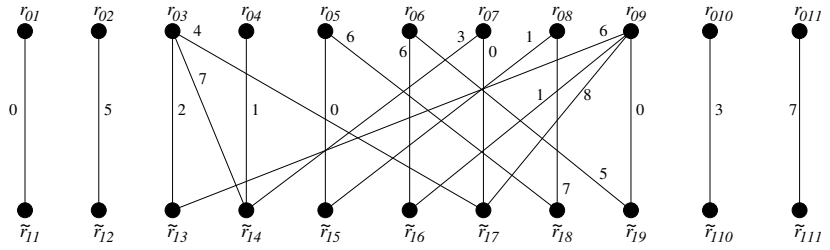
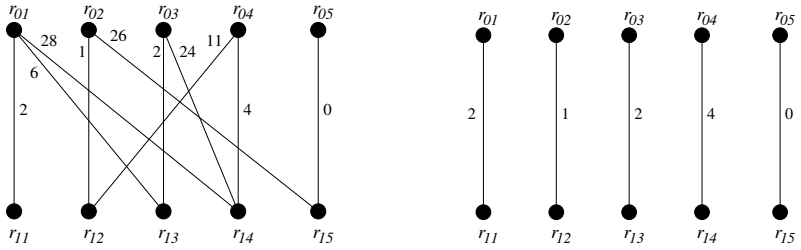


Fig. 2. The bipartite graph G_1'

Theorem 3. Let H_k be a graph obtained by removing all edges connected to r_{0i} in G_k except the one with minimum weight (degree) for all $1 \leq i \leq t$. If the degree of every node in H_k is one, then e_{ik} is equal to the weight of the edge connected to r_{0i} in H_k.

This theorem can be proved using Lemma 6 and the fact that there can not be any cycle in the graph H_k. We will give an example.

Example 6. Let $f = 25y^2z+90yz^2+93x^2y^2z+60y^4z+42z^5$. Here $t=5, n=3, \deg(f)=5$ and $p=101$. We choose the following evaluation points $\alpha_1 = 85, \alpha_2 = 96, \alpha_3 = 58$ and $b_1 = 99$. Suppose we want to construct G_2 in order to compute the degrees of the monomials in y. Suppose our degree bound is $d=40$ which is not tight. The graph G_2 and H_2 are shown in Figures 3(a) and 3(b) respectively. The graph H_2 has the correct degrees of the monomials in y.



(a) G_2 (b) H_2

Fig. 3. The bipartite graphs G_2 and H_2

Theorem 3 suggests the following optimization. In the construction of the bipartite graph G_k, connect r_{0i} to r_{kj} with degree d_{ij} *only if* there is no $\bar{d} < d_{ij}$ such that $r_{0i} \times (\frac{b_k}{\alpha_k})^{\bar{d}}$ is a root of Λ_{k+1}, i.e. the degree of the node r_{0i} in U is always one for all $1 \leq i \leq n$. If there is a perfect matching in this graph, this perfect matching is unique because this implies that the degree of each node r_{kj} in V is also one (e.g. see Figure 3(b)). If not, go back to and complete the graph G_k. This optimization makes our algorithm sensitive to the actual degree of $f(x_1, \ldots, x_n)$ in each variable.

4.3. Fast root finding over finite fields.

For our first attempt to implement our algorithm (see [16]) the most expensive part was computing the roots of the $\Lambda(z)$s. In this section we consider this problem in detail. Let p be an odd prime, $b(x) \in \mathbb{Z}_p[x]$, and $d = \deg(b)$. We suppose $d > 1$ and $b(x)$ is known to have d distinct roots and we have removed any zero root if present so that $b(0) \neq 0$. To split $b(x)$ we compute the following gcd for randomly chosen $\beta \in \mathbb{Z}_p$:

$$h(x) = \gcd((x-\beta)^{(p-1)/2} + 1, b(x)).$$

Since the polynomial $x^{p-1}-1 = (x-1)(x-2)\dots(x-(p-1))$, the polynomial $x^{(p-1)/2}-1$ is a product of half of the linear factors of $x^{p-1}-1$ and hence $(x-\beta)^{(p-1)/2}-1$ randomly shifts the linear factors so that $h(x)$ will have approximately half of the factors of $b(x)$. Of course it might have none or all of them in which case one must try another β. One then recursively splits $h(x)$ and $b(x)/h(x)$, in parallel.

This idea was presented by Rabin in [27]. It was one of the first examples of a Las Vegas algorithm. It is a beautiful algorithm. In [4] it was generalized by Cantor and Zassenhaus to a complete factorization algorithm for $\mathbb{Z}_p[x]$. Joachim von zur Gathen, in his plenary talk at ISSAC 2006 (see [7]), showed us that the main idea was already known to Gauss. We compute the gcd in two steps. First we compute

$$g(x) = (x-\beta)^{(p-1)/2-1} \mod b(x) \text{ then } h(x) = \gcd(g(x)-1, b(x)).$$

To compute $g(x)$ we use the binary powering algorithm. There are two ways to do this which we explain with an example. To compute a^{13} we think of the exponent $13 = 1101$ in binary. In the first method one computes the powers a^2, a^4, a^8 by repeated squaring in a loop and assembles a^{13} using $a^{13} = (a^1)(a^4)(a^8)$. All multiplications are reduced modulo $b(x)$. In the second method, we read the bits 1101 of 13 from left to right starting with the second bit and compute $a^{11} = (a^2)a$ then $a^{110} = (a^{11})^2$ then $a^{1101} = (a^{110})^2 a$. If $\deg(a) = d-1$ then both algorithms would cost the same. However, in our case $a(x) = x - \beta$ has degree 1 and the second method is better. We detail it here

Algorithm: Binary Powering

Input: $m \in \mathbb{N}$, $a(x), b(x) \in \mathbb{Z}_p[x]$, with $\deg(a) < \deg(b) = d$.
Output: $a(x)^m \mod b(x)$.

1: Let $B_1 B_2 \dots B_l$ be the bits of m with $B_1 = 1$, the leading bit.
2: Set $r = a$.
3: **for** $k = 2$ to l **do**
4: Set $r = r^2 \mod b$.
5: If $B_k = 1$ then set $r = a\,r \mod b$.
6: **end for**
7: **return** r.

If m is large, the multiplication and division in $r = r^2 \mod b$ will both cost $O(d^2)$ using classical arithmetic. Since $\deg(a) = 1$ the multiplication and division in $r = ar \mod b$ cost $O(d)$. For $m = (p-1)/2$ we have $l = \lfloor \log_2(p) \rfloor + 1$ thus the cost of the algorithm is $O(d^2 \log p)$ arithmetic operations in $\mathbb{Z}_p$. If we choose p to be a Fourier prime so that we can use the Fast Fourier Transform directly in $\mathbb{Z}_p$ to multiply polynomials in $\mathbb{Z}_p[x]$, the multiplication and division in $r = r^2 \mod b$ can be done in $O(d \log d)$ arithmetic

operations in $\mathbb{Z}_p$ resulting in an $O(d \log d \log p)$ algorithm. We explain here how fast division is done, its cost, in preparation for improving it.

Definition 1. Let $b(x) = b_0 + b_1 x + \cdots + b_d x^d$.
Define the *reciprocal* of b by $\hat{b} = b_d + b_{d-1} x + \cdots + b_0 x^d$.
Define the *truncation* of b by $\lfloor b \rfloor_k = b \mod x^{(k+1)} = b_0 + b_1 x + \cdots + b_k x^k$.

Let $a(x) = a_0 + a_1 x + \cdots + a_k x^k$ with $k \geq d$. Let $q(x)$ and $r(x)$ be the quotient and remainder of $a \div b$ in $\mathbb{Z}_p[x]$. Thus $a = bq + r$ with $r = 0$ or $\deg_x(r) < d$ and $\deg_x(q) = k - d$. To compute $\hat{q}$ we could expand $\hat{a}/\hat{b}$ as a power series to order x^{k-d} then take the reciprocal of the result. To do this using fast multiplication, we expand $\hat{b}^{-1}$ to degree x^{k-d} using a Newton iteration, multiply $\lfloor \hat{a} \rfloor_{k-d}$ by $\lfloor \hat{b}^{-1} \rfloor_{k-d}$ then truncate the result to degree $k - d$ to obtain $\hat{q}$. Finally we compute $a - bq$ to obtain the remainder r.

Let $M(d)$ be the cost of a multiplication in $\mathbb{Z}_p[x]$ of degree d. For $\deg b = d$ computing $\lfloor \hat{b}^{-1} \rfloor_d$ using a Newton iteration costs $3M(d) + o(d)$ (see Theorem 9.4 of [6]) arithmetic operations in $\mathbb{Z}_p$. Computing $\hat{q}$ costs one multiplication and then qb another multiplication for a total of $5M(d)$ for obtaining the remainder. Thus fast division is expensive relative to fast multiplication. We proceed to optimize the computation. Instead of counting multiplications we will count transforms (FFTs) of order n and minimize the number of FFTs of order n.

To speed up the computation we will do the entire computation in the reciprocal representation. A complication occurs when computing the remainder using $r = a - bq$ when polynomials are represented in arrays. Let $k = \deg a$, $d = \deg b$ and suppose

$$r = r_0 + r_1 x + \cdots + r_{d-1-\delta} x^l$$

where $l = d - 1 - \delta$ for some $\delta \geq 0$. In the reciprocal representation, cancellation occurs at the low order coefficients and we obtain

$$\hat{a} - \hat{b}\,\hat{q} = [\underbrace{0, 0, \ldots, 0}_{k-d+1 \text{ zeroes}}, \underbrace{0, 0, \ldots, 0}_{\delta \text{ zeroes}}, r_l, \ldots, r_1, r_0]$$

To extract $\hat{r} = r_l x^l + \cdots + r_1 x + r_0$ we first divide $\hat{a} - \hat{b}\,\hat{q}$ by x^{k-d+1} (shift the array left by $k - d + 1$). Then we need to determine l the degree of $\hat{r}$ and divide by x^δ. Note, the reason for doing this in two steps will be made clear later. We have

1: Compute $\lfloor \hat{b}^{-1} \rfloor_d$.
2: Set $\hat{r} = \hat{a}$ and $dr = 1$
3: **for** $k = 2$ to l **do**
  4: Set $\hat{r} = \hat{r}^2$ and $dr = 2dr$.
  5: If $B_k = 1$ then set $\hat{r} = \hat{a}\,\hat{r}$ and $dr = dr + 1$.
  6: **if** $dr \geq d$ **then** (divide by $\hat{b}$)
    7: Set $dq = dr - d$
    8: Set $\hat{t} = \lfloor \hat{r} \rfloor_{dq}$
    9: Set $\hat{q} = \lfloor \hat{t}\,\hat{b}^{-1} \rfloor_{dq}$
    10: Set $\hat{r} = \hat{r} - \hat{q}\,\hat{b}$
    11: Set $\hat{r} = \hat{r}/x^{dq+1}$
    12: Determine δ and dr and set $\hat{r} = \hat{r}/x^\delta$
  13: **end if**

14: **end for**
15: **return** r.

As noted by von zur Gathen and Gerhard in [6], the first optimization is to move the computation of $\hat{b}^{-1}$ out of the loop. This saves $3M(d)$ in the loop for a gain of a factor of 2 asymptotically. There are three multiplications, $\hat{r}^2$, $\hat{t}\,\hat{b}^{-1}$, and $\hat{q}\,\hat{b}$, which take, respectively, 2, 3 and 3 transforms. The reader can see that the transforms of $\hat{b}^{-1}$ and $\hat{b}$ can also be pre-computed once before the loop saving 2 out of the 8 transforms in the loop. Can we do better? Our idea is to stay in the the transform co-ordinates as much as possible. In transform co-ordinates, multiplication, addition and subtraction of polynomials are linear operations, thus $\hat{r} - \hat{q}\,\hat{b}$ can be computed in $O(n)$. Also, exact division by a monomial x^{dq+1} can be accomplished in transform co-ordinates with $O(n)$ multiplications since, for any non-zero $\omega \in \mathbb{Z}_p$, $\frac{f(x)}{x^{dq+1}}(\omega^i) = f(\omega^i)\omega^{-i(dq+1)}$. But, truncation cannot be done efficiently – we have to go out of transform co-ordinates, truncate, and come back in which costs 2 FFTs. Our improvement comes from determining l the degree of the remainder r hence $\delta = d - 1 - l$, so that we can obtain the correct the remainder $\hat{r}$ without going out of transform co-ordinates. If $\delta > 0$, then in the reciprocal representation after step 11, $\hat{r}$ will have δ leading zeroes in transform co-ordinates. After squaring $\hat{r}$ in the next iteration of the loop (and multiplying by $\hat{a}$ where $a = x - \beta$ if $B_i = 1$), $\hat{r}$ will have 2δ (provided $\beta \neq 0$) leading zeroes. We detect this and correct $\hat{r}$ when we transform $\hat{r}$ out of transform co-ordinates to truncate it in step 8. To describe the algorithm we let

$$F_\omega(b) = [b(1), b(\omega), b(\omega^2), \ldots, b(\omega^{n-1})] \in \mathbb{Z}_p^n$$

denote the Fourier transform of $b(x)$ for ω a primitive $n'th$ root of unity and F_ω^{-1} denote the inverse transform. We now present the algorithm.

Algorithm: Fast Binary Powering

Input: $m \in \mathbb{N}$, $a(x), b(x) \in \mathbb{Z}_p[x]$, with $\deg_x(a) < \deg_b(b) = d$.
We require the trailing coefficients of both a and b to be non-zero.
Output: $a(x)^m \mod b(x)$.
Comment: Variables in bold font represent polynomials in transform co-ordinates.

1: Let $B_1 B_2 \ldots B_l$ be the bits of m with $B_1 = 1$, the leading bit.
2: Compute $\lfloor \hat{b}^{-1} \rfloor_{d-1}$ using a Newton iteration and set $\hat{\mathbf{i}} = F_\omega(\lfloor \hat{b}^{-1} \rfloor_{d-1})$.
3: Set $\hat{\mathbf{b}} = F_\omega(\hat{b})$ and set $\hat{\mathbf{a}} = F_\omega(\hat{a})$.
4: Set $\hat{\mathbf{r}} = \hat{\mathbf{a}}$ and $dr = 1$.
5: **for** $k = 2$ to l **do**
6: Set $\hat{\mathbf{r}} = [\hat{\mathbf{r}}_i^2 \mod p : i = 1..n]$ and $dr = 2dr$. ——— $O(n)$
7: If $B_k = 1$ then set $\hat{\mathbf{r}} = [\hat{r}_i \hat{a}_i \mod p : i = 1..n]$ and $dr = dr + 1$. ——— $O(n)$
8: **if** $dr \geq d$ **then** (divide by $\hat{b}$)
9: Set $\hat{t} = F_\omega^{-1}(\hat{\mathbf{r}})$. ——— 1 FFT
10: Let $\hat{t} = [\underbrace{0, 0, \ldots, 0}_{2\delta \text{ zeroes}}, x, y, z, \ldots]$.
if $2\delta > 0$ **then**
11: Set $\hat{t} = [x, y, z, \ldots, 0, 0, \ldots, 0]$ and set $d_r = d_r - 2\delta$.
12: Divide $\hat{\mathbf{r}}$ by $x^{2\delta}$ in transform co-ordinates. ——— $O(n)$
13: Set $dq = dr - d$.

14: Set $\hat{\mathbf{t}} = F_\omega(\lfloor \hat{t} \rfloor_{dq})$. ——— 1 FFT
15: Set $\hat{\mathbf{t}} = [\hat{\mathbf{t}}_i \hat{\mathbf{i}}_i \mod p : i = 1..n]$. ——— $O(n)$
16: Set $\hat{\mathbf{q}} = F_\omega(\lfloor F_\omega^{-1}(\hat{\mathbf{t}}) \rfloor_{dq})$. ——— 2 FFTs
17: Set $\hat{\mathbf{r}} = [\hat{\mathbf{r}}_i - \hat{\mathbf{q}}_i \hat{\mathbf{b}}_i \mod p : i = 1..n]$. ——— $O(n)$
18: Divide $\hat{\mathbf{r}}$ by $x^{(dq+1)}$ in transform co-ordinates. ——— $O(n)$
19: Set $dr = d - 1$.
20: **end if**
21: **end for**
22: Set $\hat{r} = F_\omega^{-1}(\hat{\mathbf{r}})$.
23: Let $\hat{r} = [0, 0, \ldots, 0, r_k, \ldots, r_1, r_0, 0, 0, \ldots]$.
Set $r = r_0 + r_1 x + \ldots r_k x^k$ and **return** r.

Thus the algorithm does 4 FFTs + $O(n)$ arithmetic operations in $\mathbb{Z}_p$ in the loop instead of 8 FFTs for an overall gain of another factor of 2 asymptotically. Assuming the powers of ω have been precomputed, one FFT can be done in $\frac{n}{2} \log n$ multiplications in $\mathbb{Z}_p$, hence the cost of one iteration of the loop is $2n \log n + o(n)$.

We now determine how large n must be in terms of $\deg b = d$. The degree of $\hat{a}\hat{r}^2$ is at most $2(d-1) + 1 = 2d - 1$ thus we require $n > 2d - 1$. The degree of the quotient $dq = dr - d$ is at most $(2d-1) - d = d - 1$ hence we compute $\hat{b}^{-1}$ to degree $d - 1$. Thus we require ω to have order $n \geq 2d$.

We make one further optimization. If $d = \deg b = 600$ then we require $n \geq 2d = 1200$. Since n must be a power of 2, we need $n = 2048$. However, suppose we compute $\hat{\mathbf{r}}^2 \hat{\mathbf{i}}$ (or $\hat{\mathbf{r}}^2 \hat{\mathbf{a}} \hat{\mathbf{i}}$) in transform coordinates without truncating $\hat{\mathbf{r}}^2$ to degree dq first. We can still determine 2δ after we compute $F_\omega^{-1}(\hat{\mathbf{q}})$ and then correct $\hat{\mathbf{r}}$. Then since $\deg(\hat{a}\hat{r}^2\hat{b}^{-1}) \leq 1 + 2(d-1) + (d-1) = 3d - 2 = 1798$ we still require $n = 2048$ and we save 2 FFTs for another gain of a factor of 2 asymptotically.

In summary, we save $3M(d)$ out of $6M(d)$ by precomputing $\hat{b}^{-1}$ for a gain of one factor of 2, then we saved 4 FFTs out of 8 for a total gain of a factor of 4, and one third of the time, we can save 2 more FFTs for a total gain of a factor of 8.

5. Benchmarks

5.1. Root finding benchmarks.

We have implemented the improved binary powering algorithm in C for 31 bit primes on a 64 bit machine. We give some timing data comparing our implementation with that in Magma 2.16 and Maple 14 on an Intel Core i7 CPU running at 2.66 GHz. For the benchmarks we used the prime $p = 2114977793$. The timings in rows 1 through 4 in Table 5.1 are for computing $(x + \beta)^{(p-1)/2} \mod b(x)$ for $p = 2114977793$ a 31 bit prime with $b(x)$, monic, of degrees $d = 1000, 2000, 4000, 8000$, with coefficients chosen at random from $\mathbb{Z}_p[x]$. The timings for row 1 are for our own C implementation using classical arithmetic. The timings for row 2 are for our new improved implementation. The timings for Magma and Maple in rows 3 and 4 give some perspective. In Magma we use the `Modexp(x+beta,(p-1)/2,b);` command. In Maple we use the `Powmod(x+beta,(p-1)/2,b,x) mod p;` command.

The data in Table 1 for $d = 1000$ shows that without the improvement of the factor of 4, the fast binary powering algorithm would not break even with the classical implemen-

	d=1000	d = 2000	d = 4000	d = 8000
C classical	0.048	0.185	0.668	2.52
C new	0.011	0.023	0.047	0.097
Magma	0.056	0.129	0.310	0.610
Maple	0.800	3.11	12.21	45.0

Table 1. Timings (in CPU seconds) for polynomial arithmetic operations in $\mathbb{Z}_p[x]$.

tation until $d = 1000$. We remark that in our classical implementation for polynomial multiplication and division in $\mathbb{Z}_p[x]$, because 64 bit integer division by p is much slower than a 64 bit multiplication (on the Core i7 division takes 25 cycles and multiplication takes 3 cycles), one obtains a factor of 5 speedup by moving division by p out of the inner loop. To multiply $C = A \times B$ where A, B, C are arrays of signed int, indexed from 0, of degree da, db and $da + db$, we exploit the sign bit as follows.

```
long t,M;
M = p<<32; // M = 2^32*p;
for( i=0; i<=da+db; i++ ) C[i] = 0;
for( i=0; i<=da; i++ )
   for( j=0; j<=db; j++ )
      { t = C[i+j]; if(t>0) t -= M; C[i+j] = t+A[i]*B[j]; }
for( i=0; i<=da+db; i++ ) { C[i] %= p; if( C[i]<0 ) C[i] += p; }
```

We find that our implementation of the FFT breaks even with classical multiplication around degree 250. The timings in rows 1 through 4 of Table 2 are for computing the $d - 2$ roots of the polynomial $f = (x^d - 1)/(x^2 - 1)$ over $\mathbb{Z}_p$ for $p = 2114977793$. Maple and Magma are also using Rabin's algorithm. The timings in row 1 are for our C implementation using classical polynomial arithmetic. The timings in row 2 are for our improved fast binary powering algorithm. The timings in row Magma (Fact) are for Magma's `Factorization(f);` command (we found that Magma's `Roots(f)` command was slower). The timings in row 5 are for the `Roots(f) mod p;` command in Maple.

	d=1024	d = 2048	d = 4096	d = 8192
C classical	0.144	0.488	1.788	6.826
C new	0.096	0.247	0.618	1.449
Magma (Fact)	0.41	1.07	2.91	8.08
Maple (Roots)	1.72	6.51	24.97	95.25

Table 2. Timings (in CPU seconds) for computing Roots in $\mathbb{Z}_p[x]$.

5.2. Interpolation benchmarks.

Here, we compare the performance of our new algorithm, Zippel's algorithm and the racing algorithm of Kaltofen and Lee from [20]. We have implemented Zippel's algorithm and our new algorithm in C. We have also implemented an interface to call the interpolation routines from Maple. The racing algorithm is implemented in Maple in the ProtoBox package by Lee [20]. Since this algorithm is not coded in C, we only report (see columns labelled ProtoBox) the number of probes it makes to the black box.

We give benchmarks comparing their performance on five problem sets. The polynomials in the first four benchmarks were generated at random. The fifth set of polynomials is taken from [20]. We count the number of probes to the black box that each algorithm

takes and we measure the total CPU time for our new algorithm and Zippel's algorithm only. All timings reported are in CPU seconds and were obtained using Maple 13 on a 64 bit Intel Core i7 920 @ 2.66GHz running Linux. This is a 4 core machine. For our algorithm, we report the real time for 1 core and (in parentheses) 4 cores.

The black box in our benchmarks computes a multivariate polynomial with coefficients in $\mathbb{Z}_p$ where $p = 2114977793$ is a 31 bit prime. In all benchmarks, the black box simply evaluates the polynomial at the given evaluation point. To evaluate efficiently we compute and cache the values of $x_i^j \bmod p$ in a loop in $O(nd)$. Then we evaluate the t terms in $O(nt)$. Hence the cost of one black box probe is $O(nd+nt))$ arithmetic operations in $\mathbb{Z}_p$.

Benchmark #1

This set of problems consists of 13 multivariate polynomials in $n = 3$ variables. The i'th polynomial ($1 \le i \le 13$) is generated at random using the following Maple command:

```
> randpoly([x1,x2,x3], terms = 2^i, degree = 30) mod p;
```

The i'th polynomial will have about 2^i non-zero terms. Here $D = 30$ is the total degree hence the maximum number of terms in each polynomial is $t_{max} = \binom{n+D}{D} = 5456$. We run both the Zippel's algorithm and our new algorithm with degree bound $d = 30$. The timings and the number of probes are given in Table 3. In this table "DNF" means that the algorithm did not finish after 12 hours.

Table 3. benchmark #1: $n = 3$ and $D = 30$

i	t	New Algorithm		Zippel		ProtoBox
		Time	Probes	Time	Probes	Probes
1	2	0.00 (0.00)	13	0.00	217	20
2	4	0.00 (0.00)	25	0.00	341	39
3	8	0.00 (0.00)	49	0.00	558	79
4	16	0.00 (0.00)	97	0.01	868	156
5	32	0.00 (0.00)	193	0.01	1519	282
6	64	0.01 (0.00)	385	0.03	2573	517
7	128	0.02 (0.01)	769	0.08	4402	962
8	253	0.08 (0.03)	1519	0.21	6417	1737
9	512	0.17 (0.09)	3073	0.55	9734	3119
10	1015	0.87 (0.29)	6091	1.16	12400	5627
11	2041	3.06 (1.01)	12247	2.43	15128	DNF
12	4081	10.99 (3.71)	24487	4.56	16182	DNF
13	5430	19.02 (6.23)	32581	5.93	16430	DNF

As i increases, the polynomial f becomes denser. For $i > 6$, f has more than $\sqrt{t_{max}}$ non-zero terms. This is indicated by a horizontal line in Table 3 and also in subsequent benchmarks. The line approximately separates sparse inputs from dense inputs. The last polynomial ($i = 13$) is 99.5% dense.

The data in Table 3 shows that for sparse polynomials $1 \le i \le 6$, our new algorithm does a lot fewer probes to the black box compared to Zippel's algorithm. It also does fewer probes than the racing algorithm (ProtoBox). However, as the polynomials get denser, Zippel's algorithm has a better performance. For a completely dense polynomial with t non-zero terms, Zippel's algorithm only does $O(t)$ probes to the black box while the new algorithm does $O(nt)$ probes.

To show how effective the first optimization described in Section 4.2 is, we run both our algorithm and Zippel's algorithm on the same set of polynomials but with a bad degree bound $d = 100$. The timings and the number of probes are given in Table 4. One can see that our algorithm is unaffected by the bad degree bound; the number of probes and CPU timings are the same.

Table 4. benchmark #1: bad degree bound $d = 100$

i	t	New Algorithm		Zippel's Algorithm	
		Time	Probes	Time	Probes
1	2	0.00 (0.00)	13	0.01	707
2	4	0.00 (0.00)	25	0.01	1111
3	8	0.00 (0.00)	49	0.02	1818
4	16	0.00 (0.00)	97	0.03	2828
5	32	0.00 (0.00)	193	0.07	4949
6	64	0.01 (0.01)	385	0.14	8383
7	128	0.03 (0.01)	769	0.36	14342
8	253	0.09 (0.03)	1519	0.79	20907
9	512	0.29 (0.10)	3073	1.97	31714
10	1015	0.89 (0.31)	6091	3.97	40400
11	2041	3.08 (1.02)	12247	8.18	49288
12	4081	10.98 (3.61)	24487	15.16	52722
13	5430	18.92 (6.19)	32581	19.62	53530

Benchmark #2

In this set of benchmarks the i'th polynomial is in $n = 3$ variables and is generated at random in Maple using

```
> randpoly([x1,x2,x3], terms = 2^i, degree = 100) mod p;
```

This set of polynomials differs from the first benchmark in that the total degree of each polynomial is set to be 100 in the second set. We run both the Zippel's algorithm and our new algorithm with degree bound $d = 100$. The timings and the number of probes are given in Table 5. Comparing this table to the data in Table 3 shows that the number of probes to the black box in our new algorithm does not depend on the degree of the target polynomial.

Table 5. benchmark #2: $n = 3$ and $D = 100$

i	t	New Algorithm		Zippel		ProtoBox
		Time	Probes	Time	Probes	Probes
3	8	0.00 (0.00)	49	0.02	1919	89
4	16	0.00 (0.00)	97	0.04	3434	167
5	31	0.00 (0.00)	187	0.08	6161	320
6	64	0.01 (0.00)	385	0.19	10504	623
7	127	0.03 (0.01)	763	0.49	18887	1149
8	253	0.09 (0.03)	1519	1.38	32219	2137
9	511	0.29 (0.10)	3067	4.36	56863	4103
10	1017	0.91 (0.31)	6103	13.99	98677	7836
11	2037	3.07 (1.04)	12223	43.23	166650	DNF
12	4076	11.02 (3.61)	24457	121.68	262802	DNF
13	8147	40.68 (13.32)	48883	282.83	359863	DNF

Benchmarks #3 and #4

These sets of problems consist of 14 random multivariate polynomials in $n = 6$ variables and $n = 12$ variables all of total degree $D = 30$. The i'th polynomial will have about 2^i non-zero terms. We run both the Zippel's algorithm and our new algorithm with degree bound $d = 30$. The timings and the number of probes are given in Tables 6 and 7.

Table 6. benchmark #3: $n = 6$ and $D = 30$

i	t	New Algorithm		Zippel		ProtoBox
		Time	Probes	Time	Probes	Probes
3	8	0.00 (0.00)	97	0.01	1364	140
4	16	0.00 (0.00)	193	0.02	2511	284
5	31	0.00 (0.00)	373	0.05	4340	521
6	64	0.02 (0.01)	769	0.15	8060	995
7	127	0.06 (0.02)	1525	0.44	14601	1871
8	255	0.22 (0.07)	3061	1.51	27652	3615
9	511	0.72 (0.24)	6133	5.19	50530	6692
10	1016	2.43 (0.85)	12193	17.94	90985	12591
11	2037	8.69 (2.87)	24445	65.35	168299	DNF
12	4083	32.37 (10.6)	48997	230.60	301320	DNF
13	8151	122.5 (40.5)	97813	803.26	532549	DNF

Table 7. benchmark #4: $n = 12$ and $D = 30$

i	t	New Algorithm		Zippel		ProtoBox
		Time	Probes	Time	Probes	Probes
3	8	0.00 (0.00)	193	0.08	5053	250
4	15	0.00 (0.00)	361	0.20	10230	470
5	32	0.02 (0.01)	769	0.54	18879	962
6	63	0.06 (0.02)	1513	1.79	36735	1856
7	127	0.18 (0.05)	3049	6.10	69595	3647
8	255	0.62 (0.17)	6121	22.17	134664	7055
9	507	2.14 (0.55)	12169	83.44	259594	13440
10	1019	7.70 (1.94)	24457	316.23	498945	26077
11	2041	28.70 (7.23)	48985	1195.13	952351	DNF
12	4074	108.6 (27.2)	97777	4575.83	1841795	DNF
13	8139	421.1 (105.4)	195337	> 10000	-	DNF

Parallel benchmark.

To better assess the parallel implementation of our algorithm, table 8 reports timings for benchmark #4 for the our new algorithm with the asymptotically fast root finding algorithm running on two 6 core Intel Xeon X7460 CPUs running at 2.66GHz. We report timings and speedups for 4 and 12 cores. The data is extremely good showing a near linear speedup. For $i = 13$, if the solving were not parallelized, the maximum speedup for 12 cores would be, by Amdahl's law, $435.3/((435.3-4.2)/12+4.2) = 10.85$. However, by also parallelizing the coefficient solving step, (it is a simple observation that each coefficient can be solved for independently in $O(t)$ time) we obtain a speedup of 11.95 on 12 cores.

Table 8. Parallel speedup timing data for benchmark #4 for the new algorithm.

		1 core				4 cores		12 cores	
i	t	time	roots	solve	probe	time	(speedup)	time	(speedup)
7	127	0.218	0.01	0.00	0.12	0.062	(3.35x)	0.050	(4.2x)
8	255	0.688	0.01	0.01	0.40	0.186	(3.70x)	0.106	(6.5x)
9	507	2.33	0.05	0.02	1.53	0.603	(3.86x)	0.250	(9.3x)
10	1019	8.20	0.14	0.07	5.97	2.10	(3.90x)	0.748	(10.96x)
11	2041	30.17	0.34	0.26	23.6	7.62	(3.96x)	2.61	(11.56x)
12	4074	113.1	0.87	1.06	93.5	28.6	(3.96x)	9.90	(11.78x)
13	8139	435.3	2.25	4.20	371.7	110.5	(3.94x)	36.46	(11.95x)

Benchmark #5

In this benchmark, we compare our new algorithm and the racing algorithm on seven target polynomials (below) from [20, p. 393]. Note, f_6 is dense. The number of probes for each algorithm is reported in Table 5.2.

$$f_1(x_1,\ldots,x_9) = x_1^2x_3^3x_4x_6x_8x_9^2 + x_1x_2x_3x_4^2x_5^2x_8x_9 +$$
$$x_2x_3x_4x_5^2x_8x_9 + x_1x_3^3x_4^2x_5^2x_6^2x_7x_8^2 + x_2x_3x_4x_5^2x_6x_7x_8^2$$
$$f_2(x_1,\ldots,x_{10}) = x_1x_2^2x_4^2x_8x_9^2x_{10}^2 + x_2^2x_4x_5^2x_6x_7x_9x_{10}^2 +$$
$$x_1^2x_2x_3x_5^2x_7^2x_9^2 + x_1x_3^2x_4^2x_7^2x_9^2 + x_1^2x_3x_4x_7^2x_8^2$$
$$f_3(x_1,\ldots,x_9) = 9x_2^3x_3^3x_5^2x_6^2x_8^3x_9^3 + 9x_1^3x_2^2x_3^3x_5^2x_7^2x_8^2x_9^3 +$$
$$x_1^4x_3^4x_4^2x_5^4x_6^4x_7x_8^5x_9 + 10x_1^4x_2x_3^4x_4^4x_5^4x_7x_8^3x_9 + 12x_2^3x_4^3x_6^3x_7^2x_8^3$$
$$f_4(x_1,\ldots,x_9) = 9x_1^2x_3x_4x_6^3x_7^2x_8x_{10}^4 + 17x_1^3x_2x_5^2x_6^2x_7x_8^3x_9^4x_{10}^3 +$$
$$3x_1^3x_2^2x_6^3x_{10}^2 + 17x_2^2x_3^4x_4^2x_7^4x_8^3x_9x_{10}^3 + 10x_1x_3x_5^2x_6^2x_7^4x_8^4$$
$$f_5(x_1,\ldots,x_{50}) = \textstyle\sum_{i=1}^{i=50} x_i^{50}$$
$$f_6(x_1,\ldots,x_5) = \textstyle\sum_{i=1}^{i=5} (x_1+x_2+x_3+x_4+x_5)^i$$
$$f_7(x_1,x_2,x_3) = x_1^{20} + 2x_2 + 2x_2^2 + 2x_2^3 + 2x_2^4 + 3x_3^{20}$$

Table 9. benchmark #5.

i	n	d	$\#f_i$	New Algorithm	ProtoBox
1	9	3	5	90	126
2	10	2	5	100	124
3	9	3	5	90	133
4	9	4	5	100	133
5	50	50	50	5000	251
6	5	5	251	2510	881
7	3	20	6	36	41

The reader may observe that in all benchmarks, the number of probes our algorithm makes is exactly $2nt + 1$.

6. Conclusion

Our sparse interpolation algorithm is a modification of the Ben-Or/Tiwari algorithm [2] for polynomials over finite fields. It does a factor of between n and $2n-1$ more probes where n is the number of variables. Our benchmarks show that for sparse polynomials,

it does fewer probes to the black box than Zippel's algorithm and a comparable number to the racing algorithm of Kaltofen and Lee. Unlike Zippel's algorithm and the racing algorithm, our algorithm does not interpolate each variable sequentially and thus can easily be parallelized. Our parallel implementation using Cilk, demonstrates a very good speedup. The downside of our algorithm is that it is clearly worse than Zippel's algorithm and the racing algorithm for dense polynomials. This disadvantage is partly compensated for by the increased parallelism.

Although we presented our algorithm for interpolating over $\mathbb{Z}_p$, it also works over any finite field $GF(q)$. Furthermore, if p (or q) is too small, one can work inside a suitable extension field. We conclude with some remarks about the choice of p in applications where one may choose p.

Theorem 1 says that monomial collisions are likely when $\frac{dt^2}{2(p-1)} > \frac{1}{2}$, that is when $p - 1 < dt^2$. In our benchmarks we used 31 bit primes. Using such primes, if $d = 30$, monomial collisions will likely occur when $t > 8,460$ which means 31 bit primes are too small for applications where the number of terms t is large. The 31 bit prime limitation is a limitation of the C programming language. On a 64 bit machine, one can use 63 bit primes if one programs multiplication in $\mathbb{Z}_p$ in assembler. We are presently implementing this.

7. Acknowledgments

We would like to thank Wen-shin Lee for making the source code of the ProtoBox Maple package available to us and Joachim von zur Gathen for his suggestions for implementing Rabin's root finding algorithm and Erich Kaltofen for his helpful remarks on an early draft of our work. Also, many thanks to William Stein for letting us use his computer "geom" to generate the parallel timing data for benchmark #4.

References

[1] Holger Bast, Kurt Mehlhorn, Guido Schafer, and Hisao Tamaki. Matching algorithms are fast in sparse random graphs. *Theory of Computing Systems*, 39:3–14, 2006.

[2] M. Ben-Or and P. Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proc. of the twentieth annual ACM symposium on Theory of computing*, pages 301–309. ACM, 1988.

[3] Cilk 5.4.6 Reference Manual, `http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf`. Supercomputing Technologies Group, MIT Lab for Computer Science. `http://supertech.lcs.mit.edu/cilk`.

[4] D. G. Cantor and H. Zassenhaus. A New Algorithm for Factoring Polynomials Over Finite Fields. Mathematics of Computation **36**, 587–592, 1981.

[5] Sanchit Garg and Éric Schost. Interpolation of polynomials given by straight-line programs. *Theor. Comput. Sci.*, 410:2659–2662, 2009.

[6] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2003.

[7] J. von zur Gathen and J. Gerhard. "Who was who in polynomial factorization?" Invited talk, ISSAC 2006, Genoa, Italy, July 12, 2006.

[8] K.O. Geddes, S.R. Czapor, G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[9] M. Giesbrecht, G. Labahn and W. Lee. Symbolic-numeric sparse interpolation of multivariate polynomials. *J. Symb. Comput.*, 44:943–959, 2009.

[10] D. Grigoriev, M. Karpinski, and M. Singer. Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields. *SIAM J. Comput.* 19:1059–1063, 1990.

[11] Mark Giesbrecht and Daniel Roche. Interpolation of shifted-lacunary polynomials. *Computational Complexity*, 19:333–354, 2010.

[12] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[13] J. van der Hoven and G. Lecerf. On the bit-complexity of sparse polynomial series multiplication. Manuscript, submitted to the *Journal of Symbolic Computation*, 2010.

[14] M. A. Huang and A. J. Rao. Interpolation of sparse multivariate polynomials over large finite fields with applications. *Journal of Algorithms*, 33:204–228, 1999.

[15] S. M. M. Javadi and M. Monagan. A sparse modular gcd algorithm for polynomials over algebraic function fields. In *Proc. of ISSAC '07*, pages 187–194. ACM, 2007.

[16] Mahdi Javadi and Michael Monagan. Parallel Sparse Polynomial Interpolation over Finite Fields. In *Proceedings of PASCO '2010*, ACM Press, pp. 160–168, 2010.

[17] E. Kaltofen and Y. N. Lakshman. Improved sparse multivariate polynomial interpolation algorithms. In *Proc. of ISSAC '88*, pages 467–474. Springer-Verlag, 1989.

[18] E. Kaltofen, Y. N. Lakshman, and J.-M. Wiley. Modular rational sparse multivariate polynomial interpolation. In *Proc. of ISSAC '90*, pages 135–139. ACM, 1990.

[19] E. Kaltofen, W. Lee, and A. A. Lobo. Early termination in Ben-Or/Tiwari sparse interpolation and a hybrid of Zippel's algorithm. *Proc. of ISSAC '00*, ACM Press, 192–201, 2000.

[20] E. Kaltofen and W. Lee . Early termination in sparse interpolation algorithms. *J. Symb. Comput.*, 36(3-4):365–400, 2003.

[21] E. Kaltofen. Fifteen years after DSC and WLSS2 what parallel computations I do today. In *Proceedings of PASCO '2010*, ACM Press, pp. 10–17, 2010.

[22] E. Kaltofen. Private communication.

[23] J. L. Massey. Shift-register synthesis and bch decoding. *IEEE Trans. on Information Theory*, 15:122–127, 1969.

[24] M. Monagan J. de Kleine and A. Wittkopf. Algorithms for the non-monic case of the sparse modular gcd algorithm. In *Proc. of ISSAC '05*, pages 124–131. ACM, 2005.

[25] Rajeev Motwani. Average-case analysis of algorithms for matchings and related problems. *J. ACM*, 41:1329–1356, November 1994.

[26] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Trans. on Information Theory*, 24:106−110, 1978.

[27] Michael O. Rabin. Probabilistic algorithms in finite fields. *SIAM J. Comput*, **9**:2 273–280, 1980.

[28] Mohamed Rayes, Paul Wang, Kenneth Weber. Parallelization of The Sparse Modular GCD Algorithm for Multivariate Polynomials on Shared Memory Multiprocessors In *Proc. of ISSAC '94*, ACM Press, pp. 66–73, 1994.

[29] Jack Schwartz, Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27:701−717, 1980.

[30] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *Proc. of EUROSAM '79*, pages 216–226. Springer-Verlag, 1979.

[31] Richard Zippel. Interpolating polynomials from their values. *J. Symb. Comput.*, 9(3):375–403, 1990.