Parallel root finding over finite fields

Matthew Gibson and Michael Monagan
mdg583@gmail.com & mmonagan@cecm.sfu.ca

Abstract

Let $\mathbb{F}_q$ be a finite field with q elements and let $g(x) = \prod_{i=1}^{d}(x - \alpha_i)$ where the roots $\alpha_i \in \mathbb{F}_q$ are distinct. The classical root finding algorithm splits $g(x)$ into two factors of degree approximately $d/2$ then factors the two factors recursively. In this work we investigate how to parallelize the root finding algorithm.

The recursive factorizations may be done in parallel. However, we show that, assuming fast polynomial arithmetic, parallel speedup obtained by doing this is very limited. Therefore we parallelize the individual polynomial operations in the root finding algorithm, including HGCD, the fast Euclidean algorithm. We also consider an alternative splitting formula which we show yields a modest serial speedup but little additional parallel speedup. We have implemented our algorithms in Cilk C using asymptotically fast polynomial arithmetic. The paper includes experimental data demonstrating a modest parallel speedup.

1 Introduction

Let $\mathbb{F}_q$ be a finite field with q elements and let $f(x)$ be polynomial in $\mathbb{F}_q[x]$ of degree $d > 0$. A basic problem in computer algebra is to compute the roots of $f(x)$ that are in $\mathbb{F}_q$. Letting $M(d)$ denote the cost of multiplying two polynomials of degree d in $\mathbb{F}_q[x]$ it is classical that the roots of $f(x)$ can be computed in $O(M(d)[\log_2 d + \log_2 q])$ (see Chapter 8 of [6]).

Our interest in root finding arises from the work of Hu and Monagan in [5] where the authors developed a modular GCD algorithm which computes the greatest common divisor of two polynomials A and B in $\mathbb{Z}[x_0, x_1, \ldots, x_n]$. Let $G = \gcd(A, B)$ and let $\Gamma = \mathrm{LC}(A) \in \mathbb{Z}[x_1, \ldots, x_n]$ be the leading coefficient of A. The GCD algorithm in [5] interpolates $H = \Delta G$ where $\Delta = \Gamma/\mathrm{LC}(G)$. It does this by computing H modulo a sequence of primes then uses Chinese remaindering to recover the integer coefficients of H. The first prime q is also used to determine the support of H. The algorithm evaluates A and B at a sequence of points $\alpha_i \in \mathbb{F}_q^n$, computes monic images $g_i = \gcd(A(x_0, \alpha_i), B(x_0, \alpha_i))$ of G then interpolates the coefficients of H from scaled images $\Gamma(\alpha_i) \times g_i$ using a sparse interpolation.

The sparse interpolation used is modification of the Ben-Or Tiwari interpolation from [1] for $\mathbb{F}_q$. If $d_i = \deg_{x_i} H$ the method requires $q > \prod_{i=1}^{n} d_i$, and, because the method computes discrete logarithms in $\mathbb{F}_q$, it requires q to be a smooth prime. In the literature, this sparse interpolation approach is first described by Murao and Fujise in [11]. A description of the method may also be found in [9, 5].

In Ben-Or Tiwari sparse interpolation the most expensive step is to factor a polynomial $\lambda(z) \in \mathbb{F}_q[z]$ into linear factors. The roots of $\lambda(z)$ determine the support of the polynomial being interpolated, thus $\deg \lambda(z)$ is the number of terms in the polynomial. Since factorization of $\lambda(z)$ is the most expensive step in the sparse interpolation, we propose to parallelize it.

In [9], Kochtali, Roche and Tian write that "it is not clear how to efficiently parallelize this factorization". Their work was aimed at interpolating super sparse polynomials which have very high degree but with relatively few terms, that is, $d = \deg \lambda(z)$ is not too large. For example, their benchmarks interpolate polynomials up to degree 20^{320} for which we would need $q > 20^{320}$ and for d up to 6561. Their approach, which they call the "small primes approach", recovers these large exponents from images modulo many word-sized primes and they parallelize the algorithm on these primes. In our work we want to allow $d = 10^6$ and larger and we know that one word sized prime (63 bits) are sufficient for most GCD problems.

Section 2 presents the details of the classical root finding algorithm in $\mathbb{F}_q[z]$. The root finding algorithm is a divide and conquer algorithm. It splits $\lambda(z)$ into two factors of approximately the same degree and factors them recursively. Thus there is some natural parallelism available. However, if one uses classical quadratic polynomial arithmetic, we show that parallel speedup is limited to at most a factor of 2. The situation is better when fast polynomial arithmetic is used but parallel speedup is still very limited (see Table 2).

Splitting $\lambda(z)$ involves two operations, a modular powering operation which has no natural parallelism and a gcd computation in $\mathbb{F}_q[z]$ which we investigate in Section 3. We find that the fast Euclidean algorithm (algorithm HGCD) has natural parallelism of asymptotically a factor of 4. To inject more parallelism into the root finding algorithm, in Section 2 we also study an

alternative formula to split $\lambda(z)$ for smooth q. This formula was first suggested by Cantor and Zassenhaus in [3]. It was subsequently used by Murao and Fujise in [11] without randomization but no comparison with the classical root finding method was made. We show that if one uses this alternate formula with fast polynomial arithmetic, we get a modest serial speedup of about a factor of 2 (see Table 3) but little additional parallel speedup for $N = 2^j$ cores (see Table 4). If the number of cores N is not a power of 2 but N divides $q-1$ then we do get better parallelization. This is relevant because Intel's flagship multi-core CPUs, the Xeon E5 v4 series, come with 6,8,10,12,14,16,18,20,22 cores.

In general, it appears that parallelizing root finding is difficult. In Section 4, we compare a parallel implementation of the classical root finding with a parallel implementation of the root finding algorithm using the alternative formula and our parallel HGCD. Our parallel implementations are in Cilk C and are aimed at multi-core computers. In an attempt to inject more parallelism into the first modular powering step we also parallelize the FFT (see [10]). Our benchmarks show that this is effective for $\deg \lambda(z)$ over $50,000$. Although we obtain good results for our 16 and 20 core servers that we have access to, our approach will not scale up to hundred's of cores.

2 Parallelizing root finding

Let $\mathbb{F}_q$ be a finite field with q elements and let $f(x)$ be a non-zero polynomial in $\mathbb{F}_q[x]$. Fermat's little theorem asserts that for any element $a \in \mathbb{F}_q$, $a^q = a$ and consequently the polynomial $x^q - x$ factors as follows

$$x^q - x = \prod_{a \in \mathbb{F}_q} (x - a)$$

It follows that $g = \gcd(f(x), x^q - x)$ is the product of all the linear factors of $f(x)$. To factor g we use randomization to split $g(x)$ into two factors of approximately the same degree. For odd q, consider the factorization

$$x^q - x = x(x^{q-1} - 1) = x(x^{(q-1)/2} - 1)(x^{(q-1)/2} + 1).$$

Thus the polynomial $x^{(q-1)/2} - 1$ has half the linear factors of $x^{q-1} - 1$ and $x^{(q-1)/2} + 1$ has the other half. Suppose we pick $\alpha \in \mathbb{F}_q$ at random. Then

$$x^q - x = (x+\alpha)((x+\alpha)^{(q-1)/2} - 1)((x+\alpha)^{(q-1)/2} + 1)$$

thus the polynomial $h = \gcd(g(x), (x+\alpha)^{(q-1)/2} - 1)$ is likely to have about half the linear factors of $g(x)$ in it and the quotient h/g will have the other half. To complete the factorization of $g(x)$ one factors $h(x)$ and $g(x)/h(x)$ recursively, splitting them using different $\alpha's$ and stopping when a degree 1 or 0 factor is obtained. This leads to the following randomized algorithm.

Algorithm SPLIT
Input $g \in \mathbb{F}_q[x]$ a product of linear factors over $\mathbb{F}_q$
Output set of roots of g in $\mathbb{F}_q$

1. **if** $\deg g = 1$ **then** $g = ax + b$ so **return** $\{-b/a\}$
   if $\deg g = 0$ **then return** $\varnothing$
2. Pick $\alpha \in \mathbb{F}_q$ at random
3. $h \leftarrow \gcd(g, (x-\alpha)^{\frac{q-1}{2}} - 1)$
4. **return**(SPLIT$(h) \bigcup$ SPLIT(g/h))

This idea of splitting $g(x)$ using randomization for q a large prime first appeared in Berlekamp's factorization paper [2] from 1970. The idea is extended to a general finite field $\mathbb{F}_q$ by Rabin [12] in 1980. However, as von zur Gathen and Gerhard write in [6], "Legendre (1785) already knew the basics of the probabilistic root finding method".

Obviously the two recursive calls in Algorithm SPLIT can be done in paralllel. In order to describe the cost algorithm SPLIT we need to refer to the polynomials appearing in the computation tree of the algorithm. Let $h_0^{(0)} = g$, and let $h_0^{(0)} = h_0^{(1)} h_1^{(1)}$ where $h_0^{(1)} = \gcd(h_0^{(0)}, (x+\alpha)^{(q-1)/2} - 1)$ and $h_1^{(1)} = h_0^{(0)}/h_0^{(1)}$. At level $k > 0$ in the computation tree, if $\deg h_i^{(k-1)} > 1$ the algorithm factors

$$h_i^{(k-1)} = h_{2i}^{(k)} h_{2i+1}^{(k)} \quad \text{for} \quad i = 0, 1, \ldots, 2^{k-1}$$

using $h_{2i}^{(k)} = \gcd(h_i^{(k-1)}, (x+\alpha)^{(q-1)/2} - 1)$ where α is chosen randomly from $\mathbb{F}_q$. For $\deg h_i^{(k-1)} = d$ if we let X be the number of factors in $h_{2i}^{(k)}$ then the random variable X approximates a binomial distribution with d trials and probability $p = 0.5$. Thus $E[X] \approx d/2$ and the standard deviation of X is $\sigma \approx \sqrt{d}/2$.

Example 1 Consider the prime $p = 2^{31} - 1$ with $p - 1 = (2)(3^2)(7)(11)(31)(151)(331)$. Thus $g(x) = x^{(31)(331)} - 1 = x^{10261} - 1$ factors into linear factors over $\mathbb{F}_p$. Using $\alpha = 3$, we find that $g(x)$ splits into two factors $h_0^{(1)}(x)$ of degree 5050 and $h_1^{(1)}(x)$ of degree 5211. Using $\alpha = 5$ we find that $h_0^{(1)}(x)$ splits into $h_0^{(2)}(x)$ of degree 2531 and $h_1^{(1)}(x)$ of degree 2519 and $h_1^{(1)} = h_2^{(2)} h_3^{(2)}$ of degrees 2612 and 2599. Figure 1 depicts the computational tree. Thus we are have with four polynomials of degrees 2519, 2531, 2519 and 2599 to factor which we may factor in parallel.

To determine the parallel speedup we can obtain by simply factoring the $h_i^{(k)}$ in parallel, we need to know the relative cost of computing 2^{k-1} gcds at level k. Let $A \bmod B$ denote the remainder of $A(x) \div B(x)$ in $\mathbb{F}_q[x]$. Then

$$\gcd(g(x), (x+\alpha)^{(q-1)/2} - 1) = \gcd(g(x), w(x) - 1) \quad (1)$$

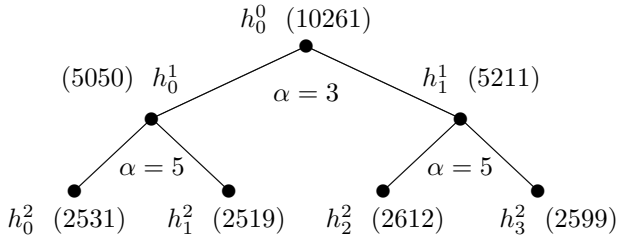

Figure 1: The top 3 levels of the computation tree for factoring $x^{10261} - 1$ over $\mathbb{F}_p$ with $p = 2^{31} - 1$. Shown in ( ) is the degree of the polynomial.

where $w(x) = (x + \alpha)^n \bmod g(x)$. The modular power $(x + 1)^{(q-1)/2} \bmod g(x)$ can be computed efficiently using the repeated squaring algorithm (Algorithm 4.8 from [6]) modulo $g(x)$. Because this computation is the most expensive part of algorithm SPLIT, and because it is a sequential bottleneck, we present the algorithm below with an example illustrating how it works.

Example 2 Let $q = 41$ so that $n = (q - 1)/2 = 21$ which is 10101 in binary. The the power $(x + \alpha)^{21} \bmod g(x)$ is computed as

$$((((x + \alpha)^2)^2(x + \alpha))^2)^2(x + \alpha)$$

where after each multiplication we reduce modulo $g(x)$ to keep the degree down.

Algorithm POWMOD(α,n,$g(x)$)
Input $\alpha \in \mathbb{F}_q$, $n > 0$ and $g(x) \in \mathbb{F}_q[x]$ of degree $d > 0$.
Output $(x + \alpha)^n \bmod g(x)$.

1 Let $n = b_l b_{l-1} \dots b_1$ be the binary representation of n with $b_l = 1$.
2 $r \leftarrow 1$.
3 **for** $i = l, l - 1, \dots, 2, 1$ **do**
  4 $r \leftarrow r^2 \bmod g$.
  5 **if** $b_i = 1$ **then** $r \leftarrow (x + \alpha)r \bmod g$.
end for.
6 **return** r.

Let $d = \deg(g)$, q be an odd prime and let $n = (q - 1)/2$. If classical quadratic polynomial multiplication and division is used, the modular power $(x + \alpha)^n \bmod g(x)$ can be computed using at most $2d^2 \log_2(q/2)$ multiplications in $\mathbb{F}_q$. If the classical quadratic Euclidean algorithm is used to compute the gcd in (1) it costs at most d^2 multiplications in $\mathbb{F}_q$. For d over about 500, fast FFT based multiplication can be used to speed up both operations. In Table 1 below $M(d)$ denotes the cost of multiplying two polynomials of degree d. If $q - 1 = 2^r s + 1$ with $2^r > 2d$ then the FFT can be used to implement fast multiplication directly in $\mathbb{F}_q$ thus $M(d) = 3F(2d) + O(d)$ where $F(n)$ is the number of arithmetic operations in $\mathbb{F}_q$ needed for an FFT. For $n = 2^k$ we have $F(n) = \frac{3}{2} n \log_2 n$ arithmetic operations in $\mathbb{F}_q$ (see Theorem 8.15 of [6]). In Table 1 below we have expressed the cost of the modular power operation and gcd operation in terms of the number of FFTs of size $2d$.

	Fast	with FFT in $\mathbb{F}_q$
$w(x)$	$O(M(d) \log_2 q)$	$c_1 F(2d) \log_2 \frac{q}{2} + O(d \log_2 q)$
$h(x)$	$O(M(d) \log_2 d)$	$c_2 F(2d) \log_2 d + O(d \log_2 d)$

Table 1: Number of arithmetic operations $\mathbb{F}_q$ for computing $w(x) = (x + \alpha)^{(q-1)/2} \bmod g(x)$ and $h(x) = \gcd(g(x), w(x) - 1)$.

In [13] Shoup showed how to do step 4 of algorithm POWMOD with only 6 FFTs of size $2d$, thus $c_1 = 6$. Let $H(d)$ be the number of number of arithmetic operations in $\mathbb{F}_q$ needed to compute the gcd of two polynomials of degree d using algorithm HGCD (see section 3). In Section 3 we show that $H(d) = 10F(d) \log d + O(d \log d)$ hence $H(d) < 5F(2d) \log d + O(d \log d)$ thus the constant $c_2 < 5$.

Let $S(l)$ be the number of arithmetic operations in $\mathbb{F}_q$ at level l in the computation tree. We are interested in the ratio $S(l)/S(l+1)$. In the best case the gcd splits $h_i^{(l)}$ of degree d into $h_{2i}^{(l+1)}$ and $h_{2i+1}^{(l+1)}$ of equal degree $d/2$. In the classical quadratic model of arithmetic

$$S(l)/S(l + 1) = \frac{2d^2 \log_2(q/2) + d^2}{2\,[2(d/2)^2 \log_2(q/2) + (d/2)^2]} \sim 2.$$

This means that the first modular power $(x+\alpha)^{(q-1)/2} \bmod g$ in algorithm SPLIT accounts for half the total work and hence the natural parallel speedup of the root finding algorithm is at most 2 on N cores.

In the fast model of polynomial arithmetic

$$S(l)/S(l+1) \sim \frac{c_1 F(2d) \log_2(q/2) + c_2 F(2d) \log_2 d}{2c_1 F(d) \log_2(q/2) + 2c_2 F(d) \log_2(d/2)}.$$

Now under the assumption that $q \gg d$, with $c_1 = 6$ and $c_2 < 5$, we have $c_1 \log_2 q > c_2 \log_2 d$ thus modular powering operation dominates the cost of all levels in the computation tree. The cost of level $l + 1$ is almost the same as level l. There is a $\log_2$ factor of the degree inside the FFT thus for $d = 2^k$ we have

$$S(l)/S(l + 1) \sim \frac{\log_2(2d)}{\log_2(d)} = \frac{k + 1}{k}.$$

If we assume that at each level the polynomials split into two factors of the same degree, then cost of the modular powers at all levels is dominated by

$$c_1(2d)\,[(k + 1) + k + \dots + 2] \log_2(q/2).$$

For $N = 2^j$ cores, if we employ 2^i cores at level i for levels $0 \le i < j$ and all N cores for levels $i \ge j$, parallel speedup is at most

$$\frac{(k+1)+k+\cdots+2}{\frac{k+1}{1}+\frac{k}{2}+\frac{k-1}{4}+\cdots+\frac{k+2-j}{N/2}+\frac{(k+1-j)+(k-j)+\cdots+2}{N}}$$

Table 2 below shows how limited the parallel speedup is for various k and j.

	number of cores $N = 2^j$				
$\deg g = 2^k$	2	4	8	16	32
2^{12}	1.75	2.55	3.16	3.50	3.65
2^{16}	1.80	2.78	3.64	4.21	4.51
2^{20}	1.83	2.94	4.04	4.84	5.31
2^{24}	1.86	3.06	4.36	5.41	6.07

Table 2: Maximum parallel speedup for N cores.

This is a best case scenario. If $h_i^{(l)}$ does not split evenly, parallel speedup is reduced. On machines with $N \neq 2^i$ cores, the work cannot be distributed evenly among the cores. To increase parallel speedup for $N = 2^j$ cores we need to parallelize the first $j-1$ levels. At each level we compute

$$h_{2i}^{(k)} = \gcd(h_i^{(k-1)}, (x+\alpha)^{(q-1)/2} \bmod h_i^{(k-1)} - 1)$$

Algorithm POWMOD does a sequence of multiplications modulo $h_i^{(k-1)}$ which cannot be parallelized. The fast Euclidean algorithm (see Chapter 11 of [6]) is also sequential. It makes two recursive calls on problems of half the original degree but the first must be completed before the second can start so there is no easy parallelism. Therefore we need to look elsewhere.

In the GCD application in [5] the prime q needs to be a smooth prime but we are otherwise free to choose q. Consider the following factorization from [3] and [11].

Lemma 2.1 *Let $n|q-1$ and let ω be a primitive n'th root of unity in $\mathbb{F}_q$. Then*

$$x^q - x = x \prod_{k=1}^{n} (x^{(q-1)/n} - \omega^k).$$

Suppose we pick $\beta \in \mathbb{F}_q$ at random. Then

$$x^q - x = (x+\beta) \prod_{k=1}^{n} ((x+\beta)^{(q-1)/n} - \omega^k). \quad (2)$$

Thus to split $g(x)$ of degree $d > 1$, we compute

$$w := (x+\beta)^{(q-1)/n} \bmod g \text{ then}$$
$$g_k := \gcd(g(x), w(x) - \omega^k) \text{ for } k = 1, 2, \ldots, n \ (*).$$

This splits $g(x)$ into n factors of degree approximately d/n. Thus if our computer has N cores and $N|q-1$ then after computing $w(x)$ we have N tasks of approximately the same size which we may run in parallel.

Example 3 Consider again the root finding problem from example 1 where $p = 2^{31} - 1$ andx $g = x^{10261} - 1$ which factors into 10261 linear factors over $\mathbb{F}_p$. Suppose our computer has $N = 6$ cores. Since $6|p-1$ we choose $n = 6$. We found $\alpha = 7$ is the smallest generator of $\mathbb{F}_p^*$ thus $\omega = \alpha^{(p-1)/n} = 1513477736$ is a primitive 6'th root of unity in $\mathbb{F}_p$. Picking $\beta = 10^6 + 1$ to "randomize the splitting", we compute $w := (x+\beta)^{(p-1)/6} \bmod g$ then $g_k := \gcd(g, w - \omega^k)$ for $k = 1, 2, \ldots, 6$. We obtain factors $g_1, g_2, \ldots, g_6$ of degree $1727, 1751, 1725, 1730, 1642, 1686$ respectively. Note, this example illustrates why we must randomize for if we use $\beta = 0$, we obtain factors $g_1, g_2, \ldots, g_6$ of degrees 0, 0, 0, 0, 0, 10261 respectively.

If we use formula (2) there is a significant savings in serial work if $n = 2^j > 2$ because we eliminate $j-1$ levels of modular power operations and gcd operations. But there is a tradeoff, namely, we must compute n gcds of degree d. Let us determine the value of j that minimizes the serial work.

Suppose $\deg g = d = 2^k \gg 1$. Then the work to compute $w(x)$ and n gcds at the first level is

$$\begin{aligned} C_{new}(j) &= c_1 F(2d) \log_2 p/n + O(d \log_2 p) \\ &+ \frac{n}{2} c_2 F(2d) \log_2 d + O(nd \log_2 d). \end{aligned}$$

The factor of 2 in $\frac{n}{2}$ is because of an optimization we will present in Section 4. The gcds $\gcd(g(x), w(x) - \omega^k)$ for $1 \le k \le n$ differ only be a constant and this can be exploited to reduce the cost by a factor of 2.

Assuming that $n \ll d$ the top j levels in the computation tree take approximately the same time thus the serial cost of the classical splitting algorithm is

$$\begin{aligned} C_{old}(j) &\approx j[c_2 F(2d) \log_2 p/2 + O(d \log_2 p) \\ &+ c_2 F(2d) \log_2 d + O(d \log_2 d)]. \end{aligned}$$

Substituting $n = 2^j$, the serial speedup is given by

$$S(j) = \frac{C_{old}(j)}{C_{new}(j)} \sim j \frac{c_1 \log_2 p/2 + c_2 \log_2 d}{c_1 \log_2 p/2^j + 2^{j-1} c_2 \log_2 d}.$$

Now maximum speedup occurs when $S'(j) = 0$. Substituting $c_1 = 6, c_2 = 5$ and considering 63 bit primes and 127 bit primes for various degrees between $2^{10} \le d \le 2^{20}$ Table 3 shows for what value of j we get maximum speedup and what that speedup is.

Notice in Table 3 that if we use the formula (2) recursively, serial speedup improves slightly as we recurse and the degree decreases.

degree	63 bit primes		127 bit primes	
d	j	speedup	j	speedup
2^{10}	3.43	2.25 ($j=3$)	4.06	2.83 ($j=4$)
2^{12}	3.28	2.15 ($j=3$)	3.89	2.67 ($j=4$)
2^{16}	3.05	1.98 ($j=3$)	3.64	2.42 ($j=4$)
2^{20}	2.90	1.85 ($j=3$)	3.44	2.24 ($j=3$)
2^{24}	2.76	1.74 ($j=3$)	3.37	2.14 ($j=3$)

Table 3: Choosing $n = 2^j$ to minimize serial work

The new formula yields a modest serial speedup. Does it also yield a parallel speedup? Let $n = 2^i$ yield the best serial cost and suppose we use the alternate on $N = 2^j$ cores for various values of j where we always do a $n = \max(2^i, 2^j)$ split. Appendix A contains a Maple program to compute the parallel speedup on $N = 2^j$ cores for various primes and degrees. The parallel speedup data is presented in Table 4 which should be compared with the data in Table 2. We are disappointed by the improvement.

			number of cores N				
$\lfloor \log_2 p \rfloor$	n	d	2	4	8	16	32
62	8	2^{12}	1.59	2.27	2.87	3.31	3.56
62	8	2^{16}	1.69	2.57	3.47	4.13	4.52
62	8	2^{20}	1.75	2.80	4.00	4.91	5.46
62	8	2^{24}	1.81	3.00	4.51	5.72	6.54
126	16	2^{12}	1.51	2.04	2.47	2.76	2.92
126	16	2^{16}	1.62	2.35	3.03	3.54	3.82
126	8	2^{20}	1.69	2.57	3.47	4.21	4.62
126	8	2^{24}	1.74	2.77	3.93	4.95	5.56

Table 4: Parallel speedup on N cores for n way split.

3 Parallelizing HGCD

A key component of the root-finding algorithm is the computation of polynomial GCD, for which we use the algorithm discussed by Knuth [15], Strassen [14], Lehmer [7] and Moenck [8] which we call the half-gcd algorithm, or HGCD. Given a field F and $A, B \in \mathbb{F}_p[x]$, one can find the greatest common divisor of A, B by constructing a Euclidean remainder sequence $(a_0 \dots a_n)$ where $a_0 = A$ and $a_1 = B$ and successive remainders are found by Euclidean division, with $q_1 \dots q_n$ the corresponding quotients:

$$\begin{aligned} a_0 &= a_1 q_1 + a_2 \\ a_1 &= a_2 q_2 + a_3 \\ &\vdots \\ a_{n-1} &= a_n q_n \end{aligned}$$

The algorithm HGCD notices that a quotient q_i can be computed using only the highest degree portion of terms of a_{i-1} and a_i. Specifically, a quotient q_i of degree k can be computed using the leading $2k$ terms of a_{i-1} and the leading k terms of a_i, see [8]. The utilization of this idea can be described with some notation (see [14]). For $a \in \mathbb{F}_p[x]$, $\deg a = m$, let $a \upharpoonright r$ be defined:

$$a \upharpoonright r = \begin{cases} 0 & \text{if } r < 0 \\ a \text{ quo } x^{m-r} & \text{if } 0 \le r \le m \\ a \times x^{r-m} & \text{if } r \ge m \end{cases}$$

Then for $a, b, a^*, b^* \in \mathbb{F}_p[x]$ with $\deg a \ge \deg b$ and $\deg a^* \ge \deg b^*$ we say that (a, b) and (a^*, b^*) coincide up to r iff $a \upharpoonright r = a^* \upharpoonright r$ and $b \upharpoonright r - (\deg a - \deg b) = b^* \upharpoonright r - (\deg a^* - \deg b^*)$. For a given Euclidean remainder sequence $(a_0, a_1, \dots a_n)$ and quotients $(q_1, q_2, \dots q_n)$ we define $\eta(k)$ to be the maximum j such that $0 \le j \le n$ and $\sum(\deg(q_1) \dots \deg(q_j)) \le k$ (see Sect. 11.1 of [6]).

Then for pairs of polynomials (a, b) and (a^*, b^*) coinciding up to $2k$ for $k \ge \deg a - \deg b \ge 0$ with the Euclidean division

$$\begin{aligned} a &= bq + r \\ a^* &= b^* q^* + r^* \end{aligned}$$

we will have $q = q^*$. Further, we will have (b, r), (b^*, r^*) coincide up to $2(k - \deg q)$ or $r = 0$ or $k - \deg q < \deg g - \deg r$ (see [6] and [14]). This leads to the following lemma which forms the basis of the HGCD algorithm.

Lemma 3.1 *For $k \in \mathbb{N}$ let (a_0, a_1) and (a_0^*, a_1^*) coincide up to $2k$, and let $h = \eta(k)$ and $h^* = \eta^*(k)$ correspond to the quotients q_i and q_i^* of the Euclidean remainder sequences of (a_0, a_1) and (a_0^*, a_1^*) respectively. Then $h = h^*$ and $q_i = q_i^*$ for $1 \le i \le h$.*

For a proof of this lemma see [6] and [14]. Given a quotient q_i we can relate consecutive remainder pairs by

$$\begin{pmatrix} a_i \\ a_{i+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \begin{pmatrix} a_{i-1} \\ a_i \end{pmatrix}$$

We may also compose these relations for $Q_i = \left(\begin{smallmatrix} 0 & 1 \\ 1 & -q_i \end{smallmatrix}\right)$ as $\left(\begin{smallmatrix} a_i \\ a_{i+1} \end{smallmatrix}\right) = Q_i Q_{i-1} \dots Q_1 \left(\begin{smallmatrix} a_0 \\ a_1 \end{smallmatrix}\right) = R_i \left(\begin{smallmatrix} a_0 \\ a_1 \end{smallmatrix}\right)$ for $R_i \in \mathbb{F}_p[x]^{2\times 2}$. Using these ideas a divide-and-conquer algorithm is constructed by choosing d a desired partial sum of quotient degrees, and acquiring the corresponding R matrix using recursion on a truncated part of the inputs. Then an intermediate remainder pair is computed and the process is repeated. We present psuedocode for algorithm HGCD so that the reader may see where to parallelize it.

A proof and analysis of this algorithm is provided by Gathen and Gerhard in [6], noting that we have here set $r_0 \upharpoonright d = r_0$ for $d \ge \deg r_0$. They show that the cost of this algorithm in additions and multiplications is bounded by $(22\,\mathrm{M}(d_0) + O(k)) \log(k)$ where $\mathrm{M}(n)$ is the

cost of a multiplication of input size n. They also show that when the degree sequence is normal, this bound becomes $(10\,\mathrm{M}(k) + O(k))\log(k)$.

Algorithm HGCD
Input: $r_0, r_1 \in \mathbb{F}_p[x]$, $d_0 = \deg r_0 \geq \deg r_1 = d_1$, and $k \in \mathbb{N}$ with $0 \leq k \leq d_0$.
Output: $R_i = Q_i \ldots Q_1 \in \mathbb{F}_p[x]^{2\times 2}$ for $i = \eta(k)$.

1. **if** $r_1 = 0$ or $k < d_0 - d_1$ **then return** $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
if $k = d_0 - d_1 = 0$ **then return** $R = \begin{pmatrix} 0 & 1 \\ 1 & -\frac{\mathrm{lc}(r_0)}{\mathrm{lc}(r_1)} \end{pmatrix}$

2. $d \leftarrow \lceil k/2 \rceil - 1$
$\tau \leftarrow \max(d_0 - 2d, 0)$
$R \leftarrow \text{HGCD}(r_0 \text{ quo } x^\tau, r_1 \text{ quo } x^\tau, d)$
$\delta_1 \leftarrow \deg(R_{2,2})$

3. $\begin{pmatrix} \tilde{r}_{j-1} \\ \tilde{r}_j \end{pmatrix} \leftarrow R \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}$

4. **if** $\tilde{r}_j = 0$ or $k < \delta_1 + \deg(\tilde{r}_{j-1}) - \deg(\tilde{r}_j)$ **return** R

5. $q_j \leftarrow \tilde{r}_{j-1} \text{ quo } \tilde{r}_j, \qquad Q_j \leftarrow \begin{pmatrix} 0 & 1 \\ 1 & -q_j \end{pmatrix}$
$\tilde{r}_{j+1} \leftarrow \tilde{r}_{j-1} \text{ rem } \tilde{r}_j$

6. $d^* \leftarrow k - \delta_1 - \deg(q_j)$
$\hat{\tau} \leftarrow \max(\deg(\tilde{r}_j) - 2d^*, 0)$
$S \leftarrow \text{HGCD}(\tilde{r}_j \text{ quo } x^{\hat{\tau}}, \tilde{r}_{j+1} \text{ quo } x^{\hat{\tau}}, d^*)$
$\delta_2 \leftarrow \deg(S_{2,2})$

7. **return** SQ_jR

To parallelize the HGCD algorithm, we will first seek an efficient serial implementation. We use the Fast Fourier Transform in steps 3 and 7. We perform the entire Matrix-Vector and Matrix-Matrix multiplications on Fourier coefficients since the image homomorphism given by the Fourier transform allows both addition and multiplication.

Let $\mathtt{F}(n)$ be the cost of a Fourier transform of degree n. At step 3 of the algorithm we will compute transforms of r_0, r_1 and the four polynomials in R, and inverse transforms of $\tilde{r}_{j-1}$ and $\tilde{r}_j$, all of the size of $\deg \tilde{r}_{j-1}$.[1] Since R is used in both step 3 and step 7, we will compute the Fourier transform of R for the larger operation and extract those coefficients corresponding to lower-order roots of unity for the smaller operation. The size of the output of step 7 is bounded by k, so that the transforms needed for step 3 are $4\,\mathtt{F}(\deg \tilde{r}_{j-1}) + 4\,\mathtt{F}(\max(\deg \tilde{r}_{j-1}, k))$. At step 7 we need a transform of q_i, as well as 4 transforms for S and 4 inverse transforms for the output. We expect q_i of small degree, so that its Fourier transform can be found in linear time using evaluation at the Fourier points. Then the FFTs needed for step 7 are $8\,\mathtt{F}(k)$.

We introduce a base case after step 1 so that when k is less than a threshold, we call a modified classical extended Euclidean algorithm with the same input and output definition as HGCD. The optimal value of the threshold of $k = 120$ was large enough to allow us to avoid the need for classical multiplication alternatives at steps 3 and 7.

3.1 Algorithm Behaviour Analysis

Since we choose $d = \lceil \frac{k}{2} \rceil - 1$ algorithm HGCD satisfies

$$\frac{k}{2} - 1 \leq d < \frac{k}{2} \leq \frac{d_0}{2} \tag{3}$$

If the algorithm doesn't return at step 4, then we also have

$$0 \leq \delta_1 \leq d < \delta_1 + \deg(q_j) \tag{4}$$

which means $d^* = k - \delta_1 - \deg(q_j) < k - d = \lfloor \frac{k}{2} \rfloor + 1$ and so $d^* \leq \frac{k}{2}$.

Let $k^{(1)}$ and $d_0^{(1)}$ be the values of k, d_0 in the recursive call of step 2, and $k^{(2)}$ and $d_0^{(2)}$ in the recursive call of step 6. Then we have $k^{(1)} = d$ and $d_0^{(1)} = \min(2d, d_0)$. By (3) we know that $2d < d_0$ so that $k^{(1)} = \frac{1}{2}d_0^{(1)}$. At step 6 consider two cases. First if HGCD is called with $k = d_0$ then since $d_0 - \delta_1 = \deg(\tilde{r}_{j-1})$ we have $d^* = d_0 - \delta_1 - \deg(q_j) = \deg(\tilde{r}_j)$ so that $d_0^{(2)} = \deg \tilde{r}_j = d^*$ and $k^{(2)} = d_0^{(2)}$. Second, if HGCD is called with $k = \frac{1}{2}d_0$ then $2d^* = \deg(\tilde{r}_j) - \delta_1 - \deg(q_j) < \deg(\tilde{r}_j)$ so that $k^{(2)} = \frac{1}{2}d_0^{(2)}$. This leads to predictable behaviour when the algorithm is called with $k = d_0$. There will be a series of second recursive calls each with $k = d_0$ for their respective inputs, while the rest of the calls in the algorithm satisfy $k = \frac{1}{2}d_0$ for their respective inputs.

Let $H(k)$ be the cost of HGCD and let us consider two scenarios. First, let $H_2(k)$ be the cost when the inputs satisfy $d_0 = 2k$. We assume $\deg q_j$ is small, so that $\deg \tilde{r}_{j-1} = d_0 - \delta_1 \approx \frac{3}{2}k$. Thus step 3 will be of cost $8\,\mathtt{F}(\frac{3}{2}k) + O(k)$ and step 7 of cost $8\,\mathtt{F}(k) + O(k)$. Letting $F(k) = k\log_2 k$ the total cost is given

$$H_2(k) = 20\,k\log_2 k + O(k) + 2H_2(k/2)$$

Solving for $H_2(k)$ with $H_2(0) = 0$ we obtain

$$H_2(k) = 10\,k\log_2^2 k + O(k\log_2 k).$$

Second, let $H_1(k)$ be the cost of HGCD when $d_0 = k$. Then $\deg \tilde{r}_{j-1} \approx \frac{1}{2}k$ and step 3 will be of cost $4\,\mathtt{F}(\frac{1}{2}k) + 4\,\mathtt{F}(k) + O(k)$, and step 7 of cost $8\,\mathtt{F}(k) + O(k)$. Thus

[1] When taking a fourier transform of size $n = 2^j$ where the input degrees are larger than $n-1$ we first reduce modulo $x^n - 1$

the cost is

$$\begin{aligned} H_1(k) &= 14\,k\log_2 k + O(k) + H_2(k/2) + H_1(k/2) \\ &= 5\,k\log_2^2 k + O(k\log_2 k) + H_1(k/2) \end{aligned}$$

Solving for $H_1(k)$ with $H_1(0) = 0$ we obtain

$$H_1(k) = 10\,k\log_2^2 k + O(k\log_2 k)$$

Then the cost in either case is

$$H(k) = 10\,F(k)\log_2 k + O(k\log_2 k)$$

3.2 Splitting at powers of 2

When using FFT based multiplications, computations may not generally be powers of two. Rather than implement the Truncated FFT, we will modify the choice of d in the algorithm at step 2 as

$$d \leftarrow 2^{\lfloor \log_2 k \rfloor} - 1$$

Then when HGCD is called with $d_0 = 2k$ and $k = 2^i - 1$ for some i, then $d = 2^{i-1} - 1$ and $d^* \le 2^{i-1} - 1$, so that d_0 and k will be just under a power of two for the majority of the algorithm. When the remainder sequence is nearly normal and $d_0 = 2k$ and $k = 2^i - 1$, then $\deg \tilde{r}_{j-1} \approx d_0 - d = 2^i + 2^{i-1} - 1 \approx \frac{3}{4}d_0$. To optimize for this we implemented a Fourier transform for input sizes $3/4$ a power of two using the cross relationships of the Truncated FFT [4] for a single level. This allows effective use of FFTs for the majority of the algorithm. Figure 2 shows timing of the HGCD algorithm with each choice for d, for GCD degree 10, r_0, r_1 of degree varying up to 1 million where coefficients are random so that the degree sequence is normal.

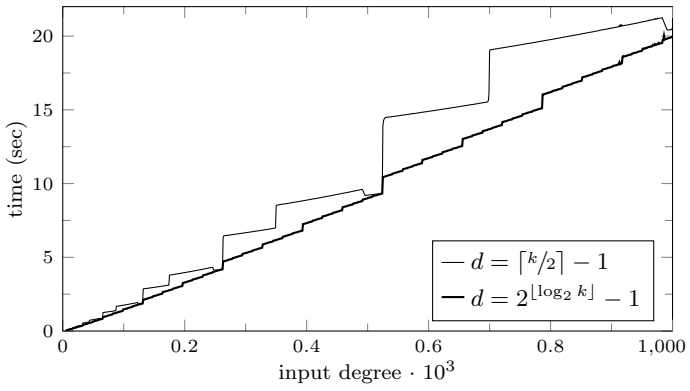

Figure 2: HGCD GCD Timings, $\deg \gcd = 10$

3.3 Paralleizing HalfGCD

Parallelization of the HGCD algorithm is difficult due to the dependency between the two recursive calls. We gain some limited parallelization in computationally intensive steps 3 and 7. For $d_0 = 2k$, at step 3 we compute six FFTs in parallel followed by two inverse transforms in parallel, all of size $\frac{3}{2}k$. At step 7 we compute four forward transforms in parallel followed by four inverse transforms in parallel, all of size k. The work excluding recursion is approximately $8\,\mathtt{F}(\frac{3}{2}k) + 8\,\mathtt{F}(k) < 20\,\mathtt{F}(k)$ while the span is $2\,\mathtt{F}(\frac{3k}{2}) + 2\,\mathtt{F}(k) < 5\,\mathtt{F}(k)$. Thus the theoretical speedup is $20/5 = 4.00$. For $d_0 = k$, at step 3 we compute six FFTs in parallel, four of which are of size k and two of size $\frac{k}{2}$, followed by two inverse transforms in parallel of size $\frac{k}{2}$. At step 7 we compute four forward transforms in parallel followed by four inverse transforms in parallel, all of size k. The work excluding recursion is approximately $4\,\mathtt{F}(k) + 4\,\mathtt{F}(\frac{k}{2}) + 8\,\mathtt{F}(k) < 14\,\mathtt{F}(k)$ while the span is $\mathtt{F}(k) + \mathtt{F}(\frac{k}{2}) + 2\,\mathtt{F}(k) < \frac{7}{2}\,\mathtt{F}(k)$. Thus the theoretical speedup is $14/3.5 = 4.00$. On four cores these theoretical speedups are 3.08 and 3.50 respectively.

4 Parallel experiments

The parallel root finding algorithm is implemented in `C` with the `CilkPlus` extension of `gcc 6.2.1`. Computations are over $\mathbb{Z}_p$ for 63-bit prime $p = 2^{49}3^25^27^2 + 1$. Our experiments were run on two Intel Xeon servers gaby and jude in the CECM at Simon Fraser University. The gaby server has two E5 2660 8 core cpus running at 2.2/3.0 GHz (base/turbo). The jude server has two E5 2680 v2 10 core cpus running at 2.8/3.6 GHz (base/turbo). Thus the maximum parallel speedup on these servers is a factor of $2.2/3.0 \times 16 = 11.7$ and $2.8/3.6 \times 20 = 15.5$ respectively. The rootfinding algorithm SWAYSPLIT is as follows:

Algorithm SWAYSPLIT
Input $g \in \mathbb{F}_q[x]$ a product of linear factors over $\mathbb{F}_q$
$S \in \mathbb{N}$ where $S | q-1$
Output set of roots of g in $\mathbb{F}_q$

1. **if** $\deg g = 1$ **then** $g = ax + b$ so **return** $\{-b/a\}$
   if $\deg g = 0$ **then return** $\varnothing$
2. Pick $\alpha \in \mathbb{Z}_q$ at random
3. $W \leftarrow (x - \alpha)^{\frac{q-1}{S}} \mod g$
4. **if** $\deg W = 0$ **then goto** 2 as $W = \omega^i$ for some i and $g_i = g$ in step 5
5. **In parallel for** $i = 1 \ldots S$ **do**
   $g_i \leftarrow \gcd(g, W - \omega^i)$
   $u_i \leftarrow$ SWAYSPLIT(g_i, S)
   end for
6. $R \leftarrow \bigcup_{i=1}^{S} u_i$
7. **if** $g(\alpha) = 0$ **then** $R \leftarrow R \cup \{\alpha\}$
8. **return** R

Step 2 is computed with the repeated squaring algorithm POWMOD of Section 2. GCDs are computed using the HGCD algorithm, but with right-hand calls of step 6 of HGCD unwrapped into a loop, avoiding one sequence of matrix-matrix multiplications and returning the GCD rather than a matrix. In the `Cilk` model of parallelism, each of N workers looks for work using a job-stealing algorithm. If $S = N$, all work will be distributed at step 5, and each core will continue the algorithm in serial. For $S > N$, parallelism will continue for additional levels.

Our implementation of SWAYSPLIT makes use of an additional serial optimization. Consider the computations of $\gcd(g, W-\omega^i)$ at step 4. We note that for each i the inputs differ only by a constant, so that according to Lemma 3.1 the first $\eta\left(\frac{1}{2}(\deg g - 1)\right)$ quotients of each remainder sequence will be equal. To use this we compute $\hat{R} = \text{HGCD}(g, W, k = \frac{1}{2}(\deg g - 1))$ after step 3 and then $\binom{\hat{r}_{j-1}}{\hat{r}_j} = \hat{R}\binom{g}{W}$ where

$$\hat{R}\begin{pmatrix} g \\ W-\omega^i \end{pmatrix} = \begin{pmatrix} \hat{r}_{j-1} \\ \hat{r}_j \end{pmatrix} - \omega^i \begin{pmatrix} \hat{R}_{1,2} \\ \hat{R}_{2,2} \end{pmatrix}$$

Then in step 5 we compute $\gcd(\hat{r}_{j-1} - \omega^i \hat{R}_{1,2}, \hat{r}_j - \omega^i \hat{R}_{2,2}) = \gcd(g, W-\omega^i)$. The savings in serial work for GCD computations is about 1/2, though parallelization is made more difficult.

We test the SWAYSPLIT algorithm on inputs of degree d constructed from random distinct linear factors. In Table 5 we investigate serial performance of the algorithm for different values of S, with the classical 2-way split represented by $S = 2$. We also show for $S = 2$ the time taken to compute the first POWMOD operation and the first GCD, showing a cost ratio between 6.8:1 to 4.7:1, reflecting the asymptotic difference of $\log_2 p$ to $\log_2 d$. As we increase S from 2 we effectively skip computations of POWMOD, computing S^i POWMODs of size d/S^i for levels $i = 0 \ldots \log_S d$. The tradeoff is increased GCD computations. We see that the optimal split for the given prime and input degrees is $S = 8$ which agrees with our theoretical estimate in Section 2, see Table 3.

We test parallelization of the classical 2-way split against the new algorithm, in Tables 6 and 7. In Table 6 we split the linear factors of g 2-ways (i.e. $S = 2$) and parallelize the 2 recursive calls on $\lceil \log_2 N \rceil$ levels, until all N threads have been assigned. In Table 7, we split the factors S ways and parallelize each of the S recursive calls on the first level, and then continue in serial, splitting S ways throughout. Parallel times are given, and parallel speedups are against the corresponding serial timings for the same choice of S, see Table 5. The results show that we have not achieved any additional parallelism for $S > 2$, though raw timings are improved due to serial speedup.

In Tables 8 to 9 we attempt to add parallelism to the POWMOD and HGCD algorithms in steps 2 and 4 of SWAYSPLIT. For the repeated squaring algorithm, we parallelize in the FFT computations. For an FFT computation of size n with t cores, at the levels $i = 0 \ldots \log_2(t)$ we break up the $\frac{n}{2}$ butterfly operations into blocks of size $\frac{n}{2tk}$, or k blocks per thread. We found $k = 4$ to give optimal parallelization for FFT sizes up to 4 million. Parallelism is also added to the HGCD algorithm as described in Section 2. The results are shown in Table 8. We see moderate success, with a speedup of 4.44 on 5 cores for input size 2 million. In Table 9 we test on jude with 20 cores using first a 5-way split followed by 4-ways splits, and we see a speedup of 12.28. In Table 10 we test on gaby with 16 cores using 16-way splits on inputs of sizes up to 8 million, with a speedup of 11.42. These are quite good in comparison to the maximum speedups on these machines described at the beginning of Section 4.

5 Concluding remarks

We have shown that the natural parallelism of the classical root-finding algorithm using a two-way split is limited. In an effort to increase parallelism we designed an efficient parallel version of the HGCD algorithm, and found that the natural parallel speedup is limited to 4.0. We also experimented with a factorization algorithm which splits more than 2 ways, which gives a serial speedup without losing parallelism. Finally, since POWMOD is a sequential bottleneck, we add some parallelism to the large FFT computations it performs. The results show that we have achieved good parallelization on 10-20 cores, though these methods won't scale up to many more cores.

References

[1] M. Ben-Or and P. Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. *Proceedings of STOC '20*, ACM Press, pp. 301–309, 1988.

[2] Elwyn Berlekamp, Factoring polynomials over large finite fields. *Mathematics of Computation*, **24**(111) 713–735, 1970.

[3] David Cantor and Hans Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, **36**(154) 587–592, 1981.

[4] Joris van der Hoeven The Truncated Fourier Transform and Applications In *Proceedings of ISSAC 2004*, ACM Press, pp. 290–296, 2004.

[5] Jiaxiong Hu and Michael Monagan. A fast parallel sparse polynomial GCD algorithm. In *Proceedings of ISSAC 2016*, ACM Press, pp. 271–278, 2016.

	$S=2$			$S=4$		$S=8$		$S=16$	
d	time	p_1	g_1	time	(×)	time	(×)	time	(×)
64000	40.26	3.14	0.46	22.09	(1.82)	17.97	(2.24)	20.31	(1.98)
128000	88.76	6.50	1.06	54.55	(1.63)	42.52	(2.09)	52.80	(1.68)
256000	191.90	13.47	2.37	107.47	(1.79)	106.60	(1.80)	100.49	(1.91)
512000	417.98	27.76	5.54	261.55	(1.60)	202.39	(2.07)	228.70	(1.83)
1024000	901.99	57.06	11.62	516.46	(1.75)	466.17	(1.93)	527.16	(1.71)
2048000	1952.02	117.06	24.95	1226.41	(1.59)	1118.90	(1.74)	1269.21	(1.54)

Table 5: Serial root-finding timings in seconds on '`gaby`'. Inputs polynomials with d random distinct linear factors. p_1 and g_1 show times for POWMOD and GCDs at the first level. Speedups are against classical $S = 2$.

	2 cores		4 cores		8 cores		16 cores	
d	time	(×)	time	(×)	time	(×)	time	(×)
64000	21.93	(1.83)	13.92	(2.88)	10.24	(3.92)	8.76	(4.58)
128000	47.81	(1.84)	29.63	(2.97)	22.01	(4.00)	18.57	(4.74)
256000	104.21	(1.84)	64.39	(2.98)	46.97	(4.09)	39.66	(4.84)
512000	226.72	(1.84)	138.01	(3.02)	100.02	(4.17)	83.95	(4.97)
1024000	486.45	(1.85)	296.33	(3.04)	213.21	(4.23)	176.83	(5.10)
2048000	1049.46	(1.87)	632.82	(3.09)	449.76	(4.35)	374.08	(5.23)

Table 6: Parallel root-finding timings with classical $S = 2$ splits on '`gaby`', with parallelism on first $\log_2(\#\text{cores})$ levels. Speedups are against corresponding serial timings, see column $S = 2$ of Table 5.

	2 cores		4 cores		8 cores		16 cores	
d	time	(×)	time	(×)	time	(×)	time	(×)
64000	21.84	(1.84)	8.18	(2.70)	5.45	(3.30)	4.61	(4.41)
128000	48.03	(1.85)	19.26	(2.83)	12.00	(3.54)	10.90	(4.84)
256000	103.92	(1.85)	39.15	(2.75)	27.13	(3.93)	21.08	(4.77)
512000	225.36	(1.85)	90.62	(2.89)	54.25	(3.73)	45.20	(5.06)
1024000	487.82	(1.85)	181.13	(2.85)	118.77	(3.93)	97.20	(5.42)
2048000	1049.29	(1.86)	414.87	(2.96)	265.78	(4.21)	213.38	(5.95)

Table 7: Parallel root-finding timings of new algorithm with $S = \#\text{cores}$ splits on '`gaby`'. Speedups are against corresponding serial timings, seen in Table 5.

	Serial			Parallel					
d	time	p_1	hgcd_1	time	(×)	p_1	(×)	hgcd_1	(×)
64000	19.88	2.26	0.33	4.95	(4.02)	1.07	(2.11)	0.20	(1.65)
128000	38.31	4.67	0.75	9.32	(4.11)	1.92	(2.43)	0.44	(1.70)
256000	92.84	9.64	1.67	22.19	(4.18)	3.50	(2.75)	0.94	(1.78)
512000	197.32	19.87	3.70	46.42	(4.25)	6.79	(2.93)	2.01	(1.84)
1024000	431.41	40.81	8.18	99.31	(4.34)	13.38	(3.05)	4.30	(1.90)
2048000	1063.93	83.67	17.97	239.63	(4.44)	27.15	(3.08)	9.21	(1.95)

Table 8: Serial and parallel root-finding timings with $S = 5$ splits and 5 cores on '`jude`'. Parallel timings include parallelization of POWMOD (p_1) and shared HalfGCD (hgcd_1) on the first level, with timings and speedups given.

	Serial			Parallel					
d	time	p_1	hgcd_1	time	(×)	p_1	(×)	hgcd_1	(×)
64000	20.12	2.26	0.33	2.21	(9.10)	0.76	(2.97)	0.21	(1.57)
128000	42.29	4.67	0.75	4.14	(10.21)	1.25	(3.74)	0.43	(1.74)
256000	98.60	9.65	1.67	8.80	(11.20)	2.22	(4.35)	0.94	(1.78)
512000	205.31	19.87	3.70	17.67	(11.62)	3.96	(5.02)	1.97	(1.88)
1024000	467.26	40.82	8.18	38.63	(12.10)	7.71	(5.29)	4.20	(1.95)
2048000	981.26	83.63	17.97	79.92	(12.28)	15.31	(5.46)	8.94	(2.01)

Table 9: Serial and parallel root-finding timings with $S = 5$ splits on the first level and $S = 4$ on remaining levels on '`jude`'. Parallel timings are on 20 cores.

	Serial			Parallel					
d	time	p_1	$hgcd_1$	time	(×)	p_1	(×)	$hgcd_1$	(×)
64000	20.33	2.95	0.45	2.31	(8.80)	0.83	(3.55)	0.27	(1.67)
128000	52.88	6.10	1.02	5.88	(8.99)	1.52	(4.01)	0.56	(1.82)
256000	100.96	12.64	2.28	10.38	(9.73)	2.91	(4.34)	1.20	(1.90)
512000	229.15	25.99	5.06	21.92	(10.45)	5.36	(4.85)	2.54	(1.99)
1024000	529.28	53.37	11.16	48.86	(10.83)	10.37	(5.15)	5.43	(2.06)
2048000	1270.34	109.24	24.52	112.48	(11.29)	20.59	(5.31)	11.59	(2.12)
4096000	2551.61	222.88	54.07	226.81	(11.25)	42.13	(5.29)	24.85	(2.18)
8192000	5663.41	453.92	117.88	496.11	(11.42)	85.55	(5.31)	52.98	(2.22)

Table 10: Serial and parallel root-finding timings with $S = 16$ splits and 16 cores on '`gaby`', degree up to 8 million.

[6] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, UK, 1999.

[7] D. H. Lehmer Euclid's Algorithm for Large Numbers *American Mathematical Monthly* **45**(4), pp. 227–233, 1938.

[8] R. T. Moenck Fast Computation of GCDs *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, pp. 142–151, 1973.

[9] Mohamed Khochtali, Daniel Roche and Xisen Tian. Parallel sparse interpolation using small primes. *Proc. PASCO 2015* 70–77, ACM Digital Library, 2015.

[10] Marshall Law and Michael Monagan. A parallel implementation for polynomial multiplication modulo a prime. *Proc. PASCO 2015* 78–86, ACM Digital Library, 2015.

[11] Hirokazu Murao and Tetsuro Fujise. Modular Algorithm for Sparse Multivariate Polynomial Interpolation and its Parallel Implementation. *J. Symb. Cmpt.* **21**:377–396, 1996.

[12] Michael Rabin. Probabilistic algorithms in finite fields. *SIAM J. Comput*, **9**:273–280, 1979.

[13] Victor Shoup. A new polynomial factorization algorithm and its implementation. *J. Symbolic Comp.* **20**(4):363–397, 1995.

[14] V. Strassen The Computational Complexity of Continued Fractions *Proceedings of the 1981 ACM Symposium*, pp. 51–67, 1981.

[15] Andrew C. Yao and Donald E. Knuth Analysis of the subtractive algorithm for greatest common divisors *Proc. Nat. Acad. Sci.* **72**(12), pp. 4720–4722, 1975.

Appendix A

Maple code to compute the parallel speedup using the n way splitting formula (2).

```
c1 := 6;
c2 := 5;

for p in [2^62,2^126] do
for d from 24 to 12 by -4 do

  for i from 1 to 6 do
    n := 2^i; # n way split
    S := 0;
    for k from d by -i to i do
      M := c1*(k+1)*log[2](p/n); # modular power
      G := c2*(k+1)*k; # fast gcd
      S := S+M+n*(G/2); # serial work
    od:
    if i=1 then minn,minS := n,S;
    elif S<minS then minn,minS := n,S;
    fi;
  od;
  printf("d=2^%d  p>2^%d  S=%d  n=%d\n",
          d,log[2](p),minS,minn);

  for j from 1 to 5 do
    N := 2^j; # number of cores
    n := max(minn,N);
    i := log[2](n);
    P := 0; # parallel time
    lev := 0;
    for k from d by -i to i do
      M := c1*(k+1)*log[2](p/n);
      G := c2*(k+1)*k;
      P := P+M/min(n^lev,N)+
           (n*G/2)/min(n^(lev+1),N);
      lev := lev+1;
    od:
    pspeedup := evalf(minS/P);
    printf(" N=%d  S/P=%8.3f\n",N,pspeedup);
  od;
od:
od;
```