Fast Parallel Multi-point Evaluation of Sparse Polynomials

Michael Monagan
Department of Mathematics
Simon Fraser University
Burnaby, B.C., Canada. V5A 1S6
mmonagan@cecm.sfu.ca

Alan Wong
Department of Mathematics
Simon Fraser University
Burnaby, B.C., Canada. V5A 1S6
cawong@sfu.ca

ABSTRACT

We present a parallel algorithm and implementation to evaluate a sparse polynomial in $\mathbb{Z}_p[x_0, \ldots, x_n]$ into many bivariate images, based on the fast multi-point evaluation technique described by van der Hoeven and Lecerf [12]. We have implemented the fast parallel algorithm in Cilk C. We present benchmarks demonstrating good parallel speedup for multi-core computers.

Our algorithm was developed with a specific application in mind, namely, the sparse polynomial GCD algorithm of Hu and Monagan [6] which requires evaluations of this form. We present benchmarks showing a large speedup for the polynomial GCD problem.

1. INTRODUCTION

We begin with a description of the motivation for and setup of our problem. Let A and B be two multivariate polynomials with integer coefficients and let $G = \gcd(A, B)$. The general idea of the GCD algorithm of [6] is to evaluate the input polynomials A, B into many univariate or bivariate images (we focus on the latter only), compute the GCD of the images with a dense algorithm, then use sparse interpolation to recover the coefficients of G. The algorithm computes G modulo a sequence of primes then reconstructs the integer coefficients of G using Chinese remaindering. So from this point forward we work over $\mathbb{Z}_p$.

Let $A = \sum_{i=1}^{s} a_i M_i(x_0, x_1, \ldots, x_n)$, where $a_i \in \mathbb{Z}_p$ and M_i is a monomial (write B similarly). We use $\#f$ to denote the number of non-zero terms of a sparse polynomial f. For convenience let us assume $\#A \geq \#B$. Let $G = \gcd(A, B) = \sum_i \sum_j g_{ij} x_0^i x_1^j$ with $g_{ij} \in \mathbb{Z}_p[x_2, \ldots, x_n]$.

First a Kronecker substitution is applied on the variables $x_2, \ldots, x_n$ to map $A \mapsto \widehat{A}(x_0, x_1, y)$ and $B \mapsto \widehat{B}(x_0, x_1, y)$. The images of G are obtained by evaluating $\widehat{A}, \widehat{B}$ at $y = \alpha^k$ and then computing their GCD, for $k = 0, 1, 2, \ldots$ where α is a primitive root of $\mathbb{Z}_p$.

Using a modified version of the Ben-Or/Tiwari method [2] to interpolate G, we require $t = 2\tau$ images where τ is the maximum of the number of terms in g_{ij} over all i, j. However, the value of τ is not known *a priori* and the strategy is to first obtain T images (T is some guess for the number of required images). Then the Berlekamp-Massey Algorithm [8] is used to generate the feedback polynomial from the images. If that polynomial stabilizes, then $T \geq t$ with high probability (see Theorem 3 of Kaltofen, Lee and Lobo [7]) so we have enough images. Otherwise we increase T and repeat until it stabilizes.

We note that in most GCD problems, the number of images that we need is very small compared to the size of the inputs. This is because $\#G$ is typically much smaller than $\#A + \#B$ and $\tau = \max(\#g_{ij})$ is much smaller than $\#G$. Because of this, Hu and Monagan [6] observed that the evaluations were the most costly part of their GCD algorithm, as the cost of the other parts of the algorithm depends on the size of G not A and B. Hence we focus on the case $s \gg t$ in the paper.

Their approach to obtain the evaluations $\gamma_k = \widehat{A}(x_0, x_1, \alpha^k)$ is as follows: Let $\widehat{A} = \sum_{i=1}^{s} a_i X_i(x_0, x_1) y^{m_i}$ where X_i is a monomial in x_0, x_1. Their algorithm first computes $\beta_i := \alpha^{m_i}$ for $1 \leq i \leq s$ using at most $D + ns$ multiplications in $\mathbb{Z}_p$ where $D = \sum_{i=2}^{n} \deg_{x_i} A$. Then computing γ_k is equivalent to the following matrix-vector multiplication:

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ \beta_1 & \beta_2 & \cdots & \beta_s \\ \vdots & \vdots & & \vdots \\ \beta_1^{T-1} & \beta_2^{T-1} & \cdots & \beta_s^{T-1} \end{bmatrix} \begin{bmatrix} a_1 X_1 \\ \vdots \\ \vdots \\ a_s X_s \end{bmatrix} = \begin{bmatrix} \gamma_0 \\ \gamma_1 \\ \vdots \\ \gamma_{T-1} \end{bmatrix} \quad (1)$$

This may be done with sT multiplications. To parallelize this on a computer with N cores, compute $\Gamma = [\beta_1^N, \beta_2^N, \ldots, \beta_s^N]$ and N vectors $E_i = [a_1\beta_1^i, a_2\beta_2^i, \ldots, a_s\beta_s^i]$ for $0 \leq i < N$ from which we obtain the first N evaluations. To obtain the next N evaluations multiply E_i by Γ for $0 \leq i < N$ in parallel to obtain $E_i = [a_1\beta_1^{i+N}, a_2\beta_2^{i+N}, \ldots, a_s\beta_s^{i+N}]$.

In this paper we present a parallel algorithm and implementation to perform these evaluations that uses the fast multi-point evaluation described by van der Hoeven and Lecerf in [12] which reduces the sT term to $O(s \log^2 T)$. In Section 2 we provide background on the fast multi-point evaluation technique for sparse polynomials. We give details on our design and implementation in Section 3, including some optimizations, our parallelization with Cilk Plus, and a note on the algorithm complexity. In Section 5 we present benchmarks comparing our implementation with the matrix method above and investigate what improvement this makes

PASCO '17 Kaiserslautern, Germany

DOI: 10.475/123_4

for the GCD problem that we described.

2. FAST MULTI-POINT EVALUATION

Our approach uses the sparse evaluation algorithm from Section 5.1 and 5.2 of [12]. Analogous formulas and algorithms for multi-point evaluation of dense polynomials can be found in [4] and [1]. In the literature, fast evaluation is presented as evaluating the polynomial down to the base field rather than evaluating to a bivariate polynomials (i.e. in our definition of $\widehat{A}$, corresponds to the case where all $X_i = 1$, and the evaluations γ_k are simply $\widehat{A}(\alpha^k)$). For compatibility we make this assumption and address the general case in Section 3.

The main idea of the fast evaluation algorithm is the observation that $\gamma_0, \ldots, \gamma_{T-1}$ are the first T coefficients in the power series expansion of the rational function

$$f(u) = \sum_{i=1}^{s} \frac{a_i}{1 - \alpha^{m_i} u}. \tag{2}$$

We illustrate this using the well-known series expansion $1 - \beta u^{-1} = 1 + \beta u + \beta^2 u^2 + \cdots + \beta^T u^T + \cdots$. We have

$$\begin{aligned} f =& \frac{a_1}{1-\beta_1 u} + \frac{a_2}{1-\beta_2 u} + \cdots + \frac{a_s}{1-\beta_s u} \\ =& a_1(1 + \beta_1 u + \beta_1^2 u^2 + \cdots) + \\ & a_2(1 + \beta_2 u + \beta_2^2 u^2 + \cdots) + \cdots + \\ & a_s(1 + \beta_s u + \beta_s^2 u^2 + \cdots) \\ =& (a_1 + \cdots + a_s) + (a_1\beta_1 + \cdots + a_s\beta_s)u + \\ & (a_1\beta_1^2 + \cdots + a_s\beta_s^2)u^2 + \cdots \\ =& \gamma_0 + \gamma_1 u + \gamma_2 u^2 + \cdots + \gamma_T u^T + \cdots \end{aligned}$$

The sum f can be split into $m = \lceil s/T \rceil$ blocks $f = B_1(u) + B_2(u) + \cdots + B_m(u)$ of size $\leq T$. To recover the evaluations of the whole polynomial after the series expansion, simply sum the coefficients over all the blocks.

A divide-and-conquer strategy (see [13]) can be applied to efficiently put the terms of each block over a common denominator, as we can split the terms of a block into two halves:

$$\begin{aligned} B_1(u) &= \sum_{i=1}^{\lceil T/2 \rceil} \frac{a_i}{1 - \alpha^{m_i} u} + \sum_{i=\lceil T/2 \rceil + 1}^{T} \frac{a_i}{1 - \alpha^{m_i} u} \\ &= B_L(u) + B_R(u) \\ &= \frac{N_L}{D_L} + \frac{N_R}{D_R} \text{ (compute recursively)} \\ &= \frac{N_L D_R + N_R D_L}{D_L D_R} = \frac{N_1}{D_1} \end{aligned} \tag{3}$$

The 3 multiplications $N_L D_R$, $N_R D_L$, and $D_L D_R$ can be computed using fast univariate multiplication. We then expand $B_i(u)$ up to $O(u^T)$ to get the coefficients using fast series inversion [13].

In total this method does $O(\lceil \frac{s}{T} \rceil M(T) \log T)$ multiplications in $\mathbb{Z}_p$ [12], where $M(T)$ denotes the cost of dense univariate polynomial multiplication of degree T. If an FFT is used and the prime p is chosen so that $2^k | (p-1)$ with $2^k > 2T$ then the FFT can run in the field $\mathbb{Z}_p$ and $M(T) \in O(T \log T)$ and we obtain a bound of $O(s \log^2 T)$ multiplications in $\mathbb{Z}_p$.

3. OUR IMPLEMENTATION

We now address the general case, where X_i can be any monomial $x_0^j x_1^k$. Note that the formulas in Section 2 are still valid here (replacing instances of a_i with $a_i X_i$), however this leads to the multiplications $N_L D_R$ and $N_R D_L$ requiring trivariate polynomial multiplication. Alternatively, we chose to sort the terms of $\widehat{A} \in \mathbb{Z}_p[y][x_0, x_1]$ into **buckets** on the monomials $x_0^j x_1^k$ as follows:

$$\widehat{A} = \sum_{j=0}^{d_0} \sum_{k=0}^{d_1} \underbrace{\left(\sum_{i=0}^{s_{jk}} a_i y^{m_i}\right)}_{\mathcal{B}_{jk} :=} x_0^j x_1^k$$

where $d_0 = \deg_{x_0}(\widehat{A}), d_1 = \deg_{x_1}(\widehat{A})$, and we denote s_{jk} as the **size of bucket** $\mathcal{B}_{jk}$ (i.e. the number of terms of $\widehat{A}$ which contain the monomial $x_0^j x_1^k$). To ease notation we relabel bucket $\mathcal{B}_{jk}$ to $\mathcal{B}_i$ where $i = j(d_1 + 1) + k$. Let $d = d_0(d_1 + 1)$ be the maximum value of i. We operate on each $\mathcal{B}_i$ as a sparse univariate evaluation problem. Observe that the computations within each bucket are independent, giving us a natural way to run the computations in parallel. Subsection 3.3 contains further details about the parallelism in our implementation.

Initialization: given α primitive in $\mathbb{Z}_p$ we precompute $\alpha^{2^k} \bmod p$ for $k = 0 \ldots \lfloor \log_2(p) \rfloor$. Let f_i be the rational function as defined by Equation (2) corresponding to $\mathcal{B}_i$. We use the precomputed powers of α to initialize f_i.

We then proceed to split $f_i = B_{i,0} + B_{i,1} + \ldots$ into blocks of size T, which raises the question of how we should choose T. Since we don't know the number of images that we need, instead of using a top-down recursive approach as outlined in Section 2, we go from the bottom up.

Starting with a relatively small guess T, we obtain T images and test for stabilization as described previously. Our strategy is to repeatedly double the value of T if we need more images. This comes from the observation that two adjacent blocks of size T can be combined into a block of size $2T$ using Equation (3). To take advantage of this we save the rational functions N_i/D_i computed at each step, and simply re-use them at the next step.

As we use FFT polynomial multiplication, we have to be somewhat careful of our starting value of T. Since we double T at every iteration, a natural choice would be $T = 2^k$. However, the denominator of a block of size 2^k has degree 2^k. An ordinary FFT of size 2^{k+1} would be wasteful. While truncated FFT algorithms are available [11], it was easier to stick with an ordinary FFT and start with $T = 63$, using $T = 63 \cdot 2^0, 63 \cdot 2^1, 63 \cdot 2^2, \ldots$ instead.

Notice that even when T is large we may still need to multiply large polynomials with small ones. We need a threshold on when to use FFT or classical multiplication. Based on our experiments, we settled on using the FFT if one of the polynomials has degree at least 250, and the other has degree at least 32. We note that tweaking these numbers did not make much of a difference to the total running time.

Figure 1 gives a demonstration of the computation for $\widehat{A} = (3y^6)x_0^2 x_1 + (y^{13} + 8y^2 + 14y^{14} + 12)x_0^3 + (5y^7 + y^4 + 11y)x_0 x_1$, $p = 17$, $\alpha = 3$. Note that the figure is somewhat misleading, as we stop at $T = 4$ which is the biggest bucket size, leading to each bucket ending up as a single block. In most cases we actually stop well before this point.

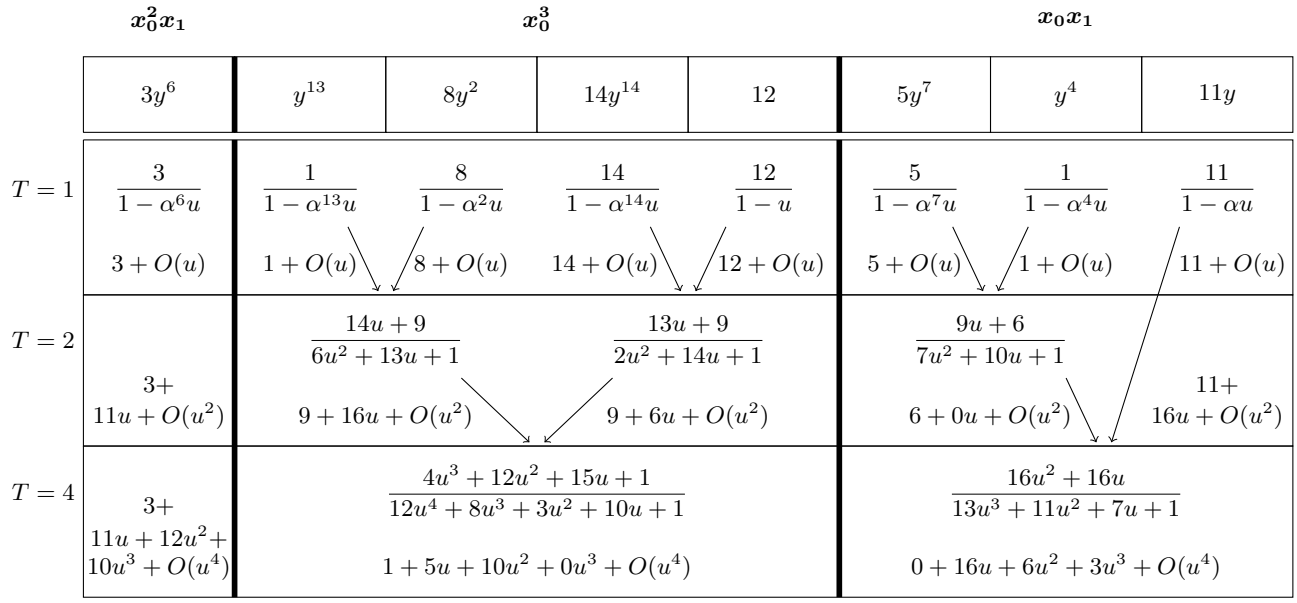

Figure 1: Computation example

3.1 Space Used

Our implementation uses 2 arrays of size s to store the rational function, one for each of the numerators and denominators. To see why this is sufficient, we observe that the block $B_{j,k} = \frac{N}{D}$ of size T corresponds to exactly T terms of $\widehat{A}$. N has degree $\leq T-1$ in u and the D has degree exactly T (both can be proved by induction). Hence N has $\leq T$ terms and D has $T+1$ terms, however as the constant term of D is always 1, it can be stored implicitly and we only explicitly store the non-constant terms in an array of size T. An example of how our data structure changes as the computation progresses is shown in Figure 2, using the computations of Figure 1.

$T=1$	N	3	1	8	14	12	5	1	11
	D	$-\alpha^6$	$-\alpha^{13}$	$-\alpha^2$	$-\alpha^{14}$	$-\alpha^0$	$-\alpha^7$	$-\alpha^4$	$-\alpha^1$
$T=2$	N	3	9	14	9	13	6	9	11
	D	2	13	6	14	2	10	7	14
$T=4$	N	3	1	15	12	4	0	16	16
	D	2	10	3	8	12	7	11	13

Figure 2: Space example

3.2 Optimizations

To compute $\frac{N}{D} = \frac{N_1D_2+N_2D_1}{D_1D_2}$ and then expand $\frac{N}{D}$ to $O(u^{2n})$ (where $2n$ is the smallest power of 2 greater than the block size T) we can do the following: multiply N_1D_2, N_2D_1 and D_1D_2 with a cost of $3M(n)$, invert D to $O(u^{2n})$ with a cost of $2M(2n)$, and multiply $N \cdot D^{-1}$ with cost $M(2n)$. The total cost is $4\frac{1}{2}M(2n)$ for this method. However, if we use the FFT to multiply polynomials it is obvious that we can save the forward transforms of D_1 and D_2 in $\frac{N_1D_2+N_2D_1}{D_1D_2}$. Also rather than doing two inverse transforms for each summand of $N_1D_2 + N_2D_1$, followed by adding the resulting polynomials, we can add the transforms and do only one inverse. Further savings are possible. In this subsection we describe and provide pseudocode for our optimizations which reduce the cost down to $2\frac{1}{6}M(2n)$.

Notation: Let $r = r_0 + r_1u + \cdots + r_{n-1}u^{n-1}$ be a polynomial of degree $< n$. To represent r we let $R = [r_0, r_1, \ldots]$ be an array of its coefficients with zeros padded. We use $FFT_n(R, \omega)$ to denote performing the FFT of order n on R with an n^{th} root of unity ω, and let $\mathcal{F}_nR$ be the resulting transform. For $A, B \in \mathbb{Z}_p^n$, let the Hadamard product of A and B be denoted by $A \otimes B = [A_0B_0, A_1B_1, \ldots, A_{n-1}B_{n-1}]$.

At several points in our algorithm we need to do the following: given $\mathcal{F}_nR$ with $deg(R) < n$, obtain $\mathcal{F}_{2n}R$ (doubling the order of the transform). The naive method is to do an inverse FFT_n^{-1} to get the original R, then do FFT_{2n} for the desired transform. We present a trick below that replaces the FFT_{2n} with an FFT_n.

Consider applying the decimation-in-frequency FFT_{2n} (as described in [13]) to $R = [r_0, r_1, \ldots, r_{n-1}, 0, \ldots, 0]$ of order $2n$. In the butterfly phase of the main call we do:

$$\begin{aligned} &\text{for } i \text{ from } 0 \text{ to } n-1 \text{ do:} \\ &\quad B_i := R_i + R_{n+i} \\ &\quad C_i := \omega^i(R_i - R_{n+i}) \end{aligned}$$

and we then return the concatenation of the recursive calls $[FFT_n(B), FFT_n(C)]$. In this case, $R_{n+i} = 0$ for all $0 \leq i < n$, so $B_i = R_i$ and $C_i = \omega^i R_i$. Hence $FFT_{2n}(R) = [\mathcal{F}_nR, \mathcal{F}_nC]$. If we are given $\mathcal{F}_nR$, it only remains to recover

R (using one FFT_n^{-1}), compute C, and do one FFT_n to obtain $\mathcal{F}_n C$.

Procedure 1 Transform doubling

1: **procedure** DO($\mathcal{F}_n R, n, \omega$) ▷ ω: $2n^{th}$ root
2: $R := \frac{1}{n} FFT_n(\mathcal{F}_n R, \omega^{-2})$
3: $C := array(n)$
4: **for** $i := 0$ **to** $n-1$ **do** $C[i] := \omega^i \cdot R[i]$ **end do**
5: $\mathcal{F}_n C := FFT_n(C, \omega^2)$
6: **return** $[\mathcal{F}_n R, \mathcal{F}_n C]$

Note that in our implementation we use an array $W = [1, \omega, \omega^2, \ldots]$ of powers of ω so that Step 4 needs only a single multiplication for each iteration, for a total of n multiplications.

We now present our pseudocode for computing and expanding $\frac{N}{D}$. We will split our algorithm into three parts: Part 1 computes the transforms of N and D, Part 2 inverts D to obtain D^{-1} to $O(u^{2n})$, and Part 3 computes $N \cdot D^{-1}$ to get the evaluations. With this optimization, we do not save the actual rational function $\frac{N}{D}$ between iterations. Instead we save the associated transforms and avoid/delay computing inverse transforms whenever possible. Hence our main input below comprises solely of transforms computed in the previous iteration.

Main Input: $\mathcal{F}_{2n}N_1$, $\mathcal{F}_{2n}N_2$, $\mathcal{F}_n D_1$, $\mathcal{F}_n D_2$, $\mathcal{F}_{2n}D_1^{-1}$, $\mathcal{F}_{2n}D_2^{-1}$, a $4n^{th}$ root of unity ω

Main Output: expansion of $\frac{N}{D}$ to $O(u^{2n})$, $\mathcal{F}_{4n}N$, $\mathcal{F}_{2n}D$, $\mathcal{F}_{4n}D^{-1}$

Part 1 computes the transforms of $N := N_1 D_2 + N_2 D_1$ and $D := D_1 D_2$. Since N, D have degree $\leq T < 2n$ we need transforms of order $2n$. As we are only given the forward transforms of D_1, D_2 to order n, we need to use the doubling trick as described above.

Part 1 Input: $\mathcal{F}_{2n}N_1$, $\mathcal{F}_{2n}N_2$, $\mathcal{F}_n D_1$, $\mathcal{F}_n D_2$, a $2n^{th}$ root of unity ω^2
Part 1 Output: $\mathcal{F}_{2n}N$, $\mathcal{F}_{2n}D$

1: Obtain the transforms of D_1, D_2 to order $2n$. Compute $\mathcal{F}_{2n}D_1 := DO(\mathcal{F}_n D_1, n, \omega^2)$ and $\mathcal{F}_{2n}D_2 := DO(\mathcal{F}_n D_2, n, \omega^2)$.
2: Compute $\mathcal{F}_{2n}N := \mathcal{F}_{2n}N_1 \otimes \mathcal{F}_{2n}D_2 + \mathcal{F}_{2n}N_2 \otimes \mathcal{F}_{2n}D_1$, and $\mathcal{F}_{2n}D := \mathcal{F}_{2n}D_1 \otimes \mathcal{F}_{2n}D_2$.
3: **return** $\mathcal{F}_{2n}N$, $\mathcal{F}_{2n}D$

To obtain $D^{-1} = b_0 + b_1 u + \cdots + b_{n-1}u^{n-1} + \cdots \bmod u^{2n}$, we use the series inversion algorithm with the middle product optimization of [5]. Recursively computing $y_k = D^{-1} \bmod u^n$ (where $n = 2^k$), the inversion algorithm uses Newton's iteration to compute $y_{k+1} = y_k + y_k(1 - Dy_k) \bmod u^{2n} = D^{-1} \bmod u^{2n}$. We make a modification to replace the recursive call using the observation that

$$\begin{aligned} D_1^{-1}D_2^{-1} + O(u^n) &= (D_1 D_2)^{-1} + O(u^n) \\ &= D^{-1} + O(u^n) \\ &= y_k + O(u^n). \end{aligned}$$

Hence we can use transforms of D_1^{-1}, D_2^{-1} (an input from previous iteration) to obtain y_k, shown in steps 4 and 5 below. Note that due to the middle product optimization, all computations in this part require only order $2n$ precision.

Part 2 Input: $\mathcal{F}_{2n}D_1^{-1}$, $\mathcal{F}_{2n}D_2^{-1}$, $\mathcal{F}_{2n}D$ (from Part 1), a $2n^{th}$ root of unity ω^2
Part 2 Output: y_{k+1}

4: Compute $\mathcal{F}_{2n}D_1^{-1}D_2^{-1} := \mathcal{F}_{2n}D_1^{-1} \otimes \mathcal{F}_{2n}D_2^{-1}$.
5: Set $y := \frac{1}{2n} FFT_{2n}(\mathcal{F}_{2n}D_1^{-1}D_2^{-1}, \omega^{-2})$ and extract the first n entries: $y_k := [y_i, i = 0, \ldots, n-1]$.
6: Compute $\mathcal{F}_{2n}y_k := FFT_{2n}(y_k, \omega^2)$.
7: Compute $\mathcal{F}_{2n}Dy_k := \mathcal{F}_{2n}D \otimes \mathcal{F}_{2n}y_k$.
8: Extract the "middle product" from the n highest order entries of $P := \frac{1}{2n} FFT_{2n}(\mathcal{F}_{2n}Dy_k, \omega^{-2})$, $MP := [P_i, i = n, \ldots, 2n-1]$.
9: Compute $\mathcal{F}_{2n}MP := FFT_{2n}(MP, \omega^2)$, negate it and multiply with the transform of y_k: $\mathcal{F}_{2n}\gamma := \mathcal{F}_{2n}y_k \otimes -\mathcal{F}_{2n}MP$.
10: To get $y_k(1 - Dy_k)$, compute $\gamma :=$ first n coefficients of $\frac{1}{2n} FFT_{2n}(\mathcal{F}_{2n}\gamma, \omega^{-2})$.
11: Combine $y_{k+1} := [y_k, \gamma]$, and **return** y_{k+1}.

Finally, Part 3 computes $N \cdot D^{-1} \bmod u^{2n}$ using the transform of N from Part 1 and $y_k = D^{-1} \bmod u^{2n}$ from Part 2. The "big" multiplication has degree $\leq 2T < 4n$ so the computations will need precision of order $4n$.

Part 3 Input: $\mathcal{F}_{2n}N$, y_{k+1}, a $4n^{th}$ root of unity ω
Part 3 Output: $N \cdot D^{-1} \bmod u^{2n}$, $\mathcal{F}_{4n}N$, $\mathcal{F}_{4n}D^{-1}$

1: Double the order of the transform of the numerator N for the big multiplication: $\mathcal{F}_{4n}N := DO(\mathcal{F}_{2n}N, 2n, \omega)$.
2: Compute $\mathcal{F}_{4n}D^{-1} := FFT_{4n}(D^{-1}, \omega)$, multiply with the transform of N: $\mathcal{F}_{4n}ND^{-1} := \mathcal{F}_{4n}N \otimes \mathcal{F}_{4n}D^{-1}$.
3: Set $Y :=$ first $2n$ entries of $\frac{1}{4n} FFT_{4n}(\mathcal{F}_{4n}ND^{-1}, \omega^{-1})$.
4: **return** $Y, \mathcal{F}_{4n}N, \mathcal{F}_{4n}D^{-1}$.

Counting the total number of transforms done (recall that $DO_n(\ldots)$ requires $2FFT_n$), we get $4FFT_n + 7FFT_{2n} + 2FFT_{4n} \leq 6\frac{1}{2}FFT_{4n} \equiv 2\frac{1}{6}M(2n)$. We get more than a factor of 2 gain compared to the $4\frac{1}{2}M(2n)$ cost by doing the computations naively. There is a space-time tradeoff for this optimization: we need five arrays of size between s and $2s$ (worst case scenario occurs if we have to pad $\approx s$ zeros) to save these transforms. So this uses at least 2.5x more space (worst case 5x) compared to the two arrays of size s described in Section 3.1.

A note on when we use this optimization: due to the space required being proportional to T rather than the bucket size, if T exceeds the size of a bucket we revert back to the original method for that bucket. Based on our data in Figure 3 of Section 5, it seems that the optimization is an improvement starting at $T \geq 504$.

3.3 Parallelization

This subsection describes how we divide up the work in each iteration for C cores. We consider the work in each iteration in two phases. The first phase is adding adjacent blocks $\frac{N}{D} = \frac{N_L}{D_L} + \frac{N_R}{D_R}$ (corresponding to Part 1 of Subsection 3.2) and the second is getting the series expansion of $\frac{N}{D}$ (Parts 2 and 3). We will parallelize the algorithm differently depending on which phase we are doing.

Notation: Suppose that our previous block size is T, and we are now combining them into $2T$-blocks. Recall that we denote the size of bucket $\mathcal{B}_i$ by s_i, and $B_{i,0}, B_{i,1}, \ldots$ denotes the T-blocks of the rational function corresponding to the terms of $\mathcal{B}_i$.

To count the number of blocks additions we have to do in bucket $\mathcal{B}_i$, observe that the number of T-blocks from the previous iteration is $\lceil s_i/T \rceil$. If that quantity is even then we can pair up all adjacent blocks, if it is odd them we pair up all but the last block. Hence, we have $b_i = \left\lfloor \frac{\lceil s_i/T \rceil}{2} \right\rfloor$ pairs and so we need to compute b_i block additions.

In total, over all the buckets we need

$$b_{total} = \sum_{i=0}^{d} b_i$$

block additions at this iteration. To parallelize this, we assign a canonical ordering to each pair of blocks that we have to add, and let each core handle $\approx \frac{b_{total}}{C}$ pairs. In an attempt to increase cache locality, we use the ordering of which the blocks are stored in our data structure.

Input: list of bucket sizes $s_0, \ldots, s_d$, list of blocks $B_{0,1}, B_{0,2}, \ldots, B_{1,0}, \ldots$, number of cores C

1: **for** $i := 0$ **to** d **do** $b_i := \lfloor \lceil s_i/T \rceil / 2 \rfloor$ **end do**
Pair up the blocks into $P = (B_L, B_R)$ and give the pairs an ordering:
2: Initialize $\sigma := 0$, which keeps track of the ordering
3: **for** $i := 0$ **to** d **do**
4: **for** $j := 0$ **to** $b_i - 1$ **do**
5: $P_{\sigma+j} := (B_{i,2j}, B_{i,2j+1})$
6: **end do**
7: $\sigma \mathrel{+}= b_i$
8: **end do**
9: Distribute $P_0, \ldots, P_{b_{total}}$ among C cores, each core will do the block addition for the pairs that they are given.

In the second phase, instead of parallelizing on all the blocks, we will just parallelize on the buckets. Our idea is to form C subsets of buckets which require roughly equal work (this will be estimated based on the bucket size). When we "evaluate" a bucket $\mathcal{B}_i$, we are referring to obtaining the T evaluations by series expansion.

3.4 Complexity

Suppose that the total number of evaluations needed is t. Recall that the matrix method will do $O(st + nd + ns)$ multiplications in $\mathbb{Z}_p$. For convenience we will assume that the fast algorithm computes in powers of two for the complexity analysis. So each iteration computes $T_k = 2^k$ evaluations for $k = 0, 1, \ldots$ and let K be the smallest integer s.t. $T_k = 2^K \geq t$ (i.e. K is the number of iterations to get t images). Note that $K = \lceil \log_2(t) \rceil$. The cost depends on how the terms of $\widehat{A}$ are distributed over the buckets, since if we have very small buckets $\mathcal{B}_i$ with size $s_i < c\log^2 t$ (for some constant c), then using fast evaluation and FFT multiplication is slower than using the matrix method. These small buckets can be identified before the algorithm starts and omitted from the fast evaluation, so we assume that all block sizes s_i are at least $c\log^2 t$.

Input: list of buckets $\mathcal{B}_0, \ldots, \mathcal{B}_d$ and their sizes $s_0, \ldots, s_d$, number of cores C

Sort the buckets by their sizes, rounding up to the next power of 2:
1: Initialize R as an array of empty sets
2: **for** $i := 0$ **to** d **do**
3: $s := \lceil \log_2(s_i) \rceil$
4: $R_s := R_s \cup i$
5: **end do**

6: **while** not all buckets have been evaluated **do**
7: Let h be the highest index j s.t. R_j is non-empty.
8: Divide R_h into C subsets of size $\lceil |R_h|/C \rceil$.
9: If some subsets are not full, greedily add buckets from $R_{h-1}, R_{h-2}, \ldots$, using the approximation that a bucket in R_j is worth two buckets in R_{j-1}.
10: Send off one subset to each core to be evaluated.
11: Remove all evaluated buckets from R.
12: **end do**

We first count the number of multiplications done in an individual bucket $\mathcal{B}_i$ with size s_i. At the k^{th} iteration, we get $T_k = 2^k$ evaluations for $\lceil \frac{s_i}{T_k} \rceil$ blocks. Each block requires $M(T_k)$ multiplications to add the rational functions and get the series expansion. To determine the cost of for $\mathcal{B}_i$ we consider two cases:

- **Case 1** $s_i \geq t$ and $M(T_k) = O(T_k \log T_k)$ (using FFT multiplication):

$$\begin{aligned}
\sum_{k=0}^{K} O\left(\left\lceil \frac{s_i}{T_k} \right\rceil M(T_k)\right) &= \sum_{k=0}^{K} O\left(\left\lceil \frac{s_i}{T_k} \right\rceil T_k \log T_k\right) \\
&= \sum_{k=0}^{K} O(s_i \log T_k) \\
&\subseteq \sum_{k=0}^{K} O(s_i \log t) \\
&= O(\log t \cdot s_i \log t) \\
&= O(s_i \log^2 t).
\end{aligned}$$

- **Case 2** $s_i < t$ and $M(T_K) = O(T_k \log T_k)$:

$$\begin{aligned}
\sum_{k=0}^{K} O\left(\left\lceil \frac{s_i}{T_k} \right\rceil M(T_k)\right) &\subseteq \sum_{k=0}^{K} O(M(T_k)) \\
&\subseteq O(t \log^2 t)
\end{aligned}$$

If we assume that there are a negligible number of buckets that fall into Case 2, then we get a total cost of $O(s\log^2 t + sn\log d)$, where $O(sn\log d)$ is the cost of initializing the rational function ($n\log d$ multiplications to compute α^m where $m < d^{n-2}$). We remark that in certain contexts we are allowed some flexibility in choosing the special variables x_0 and x_1, so that swapping x_0, x_1 with two other variables may yield more buckets in Case 1.

4. IMPLEMENTATION

We implemented two versions of our algorithm in C, one using 64-bit integer types and 63-bit primes, the second using the 128-bit integer `__int128_t` and 127-bit primes. We use primes with one less bit than the maximum size for a couple of reasons. The main consideration is that we want to reduce the nubmer of division operations performed in our implementation. A division of a 128-bit integer by a 64-bit integer is very expensive as it costs 66.602 CPU cycles, compared to 2.665 cycles for multiplying two 64-bit integers. To compute the linear combination $c = a_1b_1 + a_2b_2 + \ldots + a_nb_n \mod p$ (where p is 63-bit), instead of dividing though by p after every sum, we can accumulate the value of $-c \mod p$ with an 128-bit integer, and do a single division at the end to recover the value of c (the case for a 127-bit prime is analogous). Also, by leaving an unused bit we can represent every number mod p with signed integers and use the sign bit to do addition and subtraction mod p without overflow.

While the above handles the case of linear combinations, the majority of the cost of our algorithm comes from computing $a \times b \mod p$ for many different values of a and b, so we still need to optimize this operation. The division algorithm of Möller and Granlund [9] uses an approximate reciprocal to transform a division operation into a multiplication along with some relatively cheap adjustments. Using the implementation of this division algorithm by Roman Pearce, we first precompute the reciprocal of p, and reuse it whenever possible. Using a precomputed reciprocal reduces the cost to only 6.106 CPU cycles. As we compute $a \times b \mod p$ many times in our algorithm for the same prime p, we see a large gain when using this optimization.

Our parallel code uses the `-fcilkplus` option in the `gcc` compiler, which enables Intel Cilk Plus. We make use of the options `cilk_spawn`, which allows Cilk to branch off the execution of a function in parallel, and `cilk_sync`, which signals the main thread to wait until all preceding spawned jobs have been completed. Our memory management with respect to the parallel jobs can be illustrated with the following example; to run N tasks in parallel for N cores we do:

```
for( i=0; i<N; i++ )
{
    spawn Task(Input[i],  // space for inputs
               Output[i], // space for outputs
               Temp[i]);  // working storage
}
sync;
```

So our model is that we always pre-allocate all heap space needed for outputs and working storage, for each parallel task. For example, neither Task(...) nor its subroutines allocates heap space.

5. BENCHMARKS

We generated random sparse polynomials in $n = 9$ variables, with degree at most 10 in $x_1, \ldots, x_n$ and total degree at most 60. Each timing represents running our algorithm on a polynomial with s terms, until we obtain $T \geq t$ images for some test parameter t (for the fast times $T \in \{63, 126, \ldots\}$).

All timings were made on the gaby server in the CECM at Simon Fraser University. This machine has two Intel Xeon E-2660 8 core CPUs running at 3.0 GHz on one core and 2.2 GHz on 8 cores. The maximum theoretical parallel speedup is $11.73 = 2.2/3.0 \times 16$.

Figure 3 shows the timings for the matrix method and the fast method (with and without the transform optimizations from Subsection 3.2). The timing that is fastest for each t is shown in bold. The fast algorithm with the optimization is clearly superior when we need at least 504 evaluations. The Diff column in the table shows the difference in the timings compared to the entry above (i.e. the extra time it takes for the algorithm to get the next set of evaluations). We observe that, starting at $t = 504$ the optimized version takes about half as much time as the non-optimized version to get the next set of evaluations, which matches the factor of 2 gain in the analysis of Section 3.2.

s	t	Matrix	Fast	Diff	FastOpt	Diff
10^7	63	**1.261**	1.656	-	2.248	-
10^7	126	**2.149**	2.367	0.711	3.181	0.933
10^7	252	3.900	**3.568**	1.201	4.140	0.959
10^7	504	7.165	5.681	2.113	**5.177**	0.977
10^7	1008	15.678	7.870	2.189	**6.284**	1.107

Figure 3: Break-even for fast implementations

We collected data for a larger input size and larger t, shown in Figure 4. The timings for both 1 core and 16 cores are presented to show the parallel speedup that we get.

The distribution of bucket sizes for the polynomials are relevant since fast evaluation is most effective when t is strictly smaller than each bucket. The bucket sizes for the above polynomials with 10^8 terms were fairly large and uniform, as they ranged from 51263, 72018, 86445, ..., 1876418, 1899002, 2131494, and we have 114 of the 121 buckets with size between $2^{17} = 131072$ and 2^{21}. To test our algorithm on polynomials with a much different distribution of bucket size, we homogenize the above polynomials on x_0 (this operation also has a specific application to computing GCDs). The homogenization increases the number of buckets to 554. The bucket sizes now range from $1, 1, 1, \ldots, 818007, 831529, 833626$, and we have 316 buckets with size under 10^5. Due to the presence of these small buckets, we observe below that fast evaluation does not perform as well compared to the non-homogenized polynomials, for the cases of $t = 10^5$ and $t = 10^6$. Figure 5 reports our data for using the fast method to evaluate these polynomials (the data for the matrix method omitted as it is clear that homogenizing does not affect the dominating term $O(st)$).

5.1 Polynomial GCD Benchmarks

To show the improvement that this fast evaluation algorithm makes to the GCD algorithm of Hu and Monagan [6], we display timings for the GCD algorithm using our fast evaluation implementation in Figure 6. The timings in columns Fast and Matrix are for 16 cores. The GCD algorithm of Hu and Monagan is in Cilk C.

We randomly generated polynomials $\overline{A}, \overline{B}, G$ and constructed $A := \overline{A} \cdot G$, $B := \overline{B} \cdot G$. Each of $G, \overline{A}, \overline{B}$ has 9 variables, degree in each variable at most 20, and total degree at most 60.

The timings for the parameters dubbed as the "benchmark problem" in Hu and Monagan [6] where ($\#A = 10^6$, $\#G = 10^4$) are shown in bold. Our timing of 0.6s is faster than the

		Matrix			FastOpt		
s	t	1 core	16 cores	Speedup	1 core	16 cores	Speedup
10^7	10^3	90.73	7.06	12.9x	40.81	3.39	12.0x
10^7	10^4	898.87	66.40	13.5x	77.18	6.50	11.9x
10^7	10^5	8804.57	682.53	12.9x	130.03	11.70	11.1x
10^8	10^4	8149.87	669.49	12.2x	671.35	57.38	11.7x
10^8	10^5	-	6608.95	-	974.70	84.11	11.6x
10^8	10^6	-	-	-	1497.60	131.95	11.3x

Figure 4: Timings (in seconds) for $T > t$ evaluations of A with s terms

		FastOpt-Homogeneous		
s	t	1 core	16 cores	Speedup
10^7	10^3	41.13	3.26	12.6x
10^7	10^4	84.50	6.92	12.2x
10^7	10^5	221.90	21.84	10.2x
10^8	10^4	679.35	54.85	12.4x
10^8	10^5	1055.49	88.78	11.9x
10^8	10^6	2462.27	259.83	9.5x

Figure 5: Timings (in seconds) for homogeneous A

timing of 4.47s reported in [6] even though we are running on the same machine. The reason is that here we evaluate to bivariate images which reduced t from 2396 evaluations to 264.

As can be seen in Figure 6, most of the time is spent in evaluation. For the largest case $\#A = 10^8, \#G = 10^7$ terms, most of the time is in computing the roots of the feedback polynomial which has degree $\tau = t/2$. The algorithm being used for root finding is the randomized root finding algorithm of Berlekamp [3]. See also Rabin [10].

Shown also for comparison are timings for the GCD algorithm in Maple 2016 and Magma 2.22-5. Maple and Magma are both using Zippel's sparse modular GCD algorithm from [14]. Both implementations are serial implementations in C.

6. REFERENCES

[1] A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley Pub. Co., 1974.

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

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

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

[5] G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm i. *Appl. Algebra Eng., Commun. Comput.*, 14(6):415–438, Mar. 2004.

[6] J. Hu and M. Monagan. A fast parallel sparse polynomial gcd algorithm. In *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '16, pages 271–278. ACM, 2016.

[7] E. Kaltofen, W. shin Lee, and A. Lobo. Early termination in ben-or/tiwari sparse interpolation and a hybrid of zippel's algorithm. In *Proceedings of ISSAC 2000*, pages 192–201. ACM Press, 2000.

[8] J. Massey. Shift-register synthesis and bch decoding. *IEEE transactions on Information Theory*, 15(1):122–127, 1969.

[9] N. Moller and T. Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60(2):165–175, 2011.

[10] M. Rabin. Probabilistic algorithms in finite fields. *SIAM Journal of Computing*, 9:273 – 280, 1979.

[11] J. van der Hoeven. The truncated fourier transform and applications. In *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 290–296. ACM, 2004.

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

[13] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2nd edition, 2003.

[14] R. Zippel. Probabilistic algorithms for sparse polynomials. In *Proceedings of EUROSAM '79*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226. Springer-Verlag, 1979.

$\#A$	$\#G$	t	Fast	(eval)	Matrix	(eval)	Maple	Magma
10^5	10^3	36	0.1	(76%)	0.1	(55%)	210.9	60.2
10^6	10^3	40	0.5	(88%)	0.2	(66%)	2135.9	207.6
10^6	**10^4**	**264**	**0.8**	(82%)	**0.6**	(74%)	**22111.6**	**1611.5**
10^6	10^5	2336	4.9	(45%)	6.1	(57%)	-	876.9
10^7	10^4	256	5.8	(90%)	4.5	(88%)	-	8334.9
10^7	10^5	2334	13.5	(77%)	36.1	(91%)	-	72341.0
10^7	10^6	24214	91.1	(32%)	395.7	(85%)	-	-
10^8	10^4	246	46.2	(89%)	45.8	(91%)	-	-
10^8	10^5	2328	96.3	(92%)	369.2	(98%)	-	-
10^8	10^6	24214	214.9	(69%)	3691.1	(98%)	-	-
10^8	10^7	242574	3058.1	(11%)	39643.0	(93%)	-	-

Figure 6: Gcd Timings (in seconds) with #A = #B