Maple Transactions

Implementing Kaltofen and Yagati's fast transposed Vandermonde solver

HYUKHO KWON and MICHAEL MONAGAN*, Department of Mathematics, Simon Fraser University, Canada

CCS Concepts: • **Computing methodologies** → **Algebraic algorithms**.

Additional Key Words and Phrases: Transposed Vandermonde Systems, Fast Algorithms

Recommended Reference Format:
Hyukho Kwon and Michael Monagan. 2024. Implementing Kaltofen and Yagati's fast transposed Vandermonde solver. 1, 1 (November 2024), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

We present a C implementation of Kaltofen and Yagati's fast transposed Vandermonde solver from [6] over the finite field $\mathbb{Z}_p$ for p a prime with at most 63 bits. For comparison, we have also implemented Kaltofen and Yagati's algorithm in Maple and Zippel's algorithm from [9] in C. Our goal is to determine how much faster an optimized C implementation is than an implementation that just uses the fast polynomial multiplications and divisions in $\mathbb{Z}_p[x]$ provided by the system (Maple).

The motivation for our work is the black-box multivariate polynomial factorization algorithm of Chen and Monagan [3] which needs to solve many transposed Vandermonde systems. In [3] the authors factor the determinant of the 16 by 16 symmetric Töplitz matrix. About 3/4 of the total factorization time is spent solving transposed Vandermonde systems, the largest of which is 127,690 by 127,690.

Let F be a field and $a(x) = \sum_{j=1}^{n} a_j x^{e_j}$ be an unknown polynomial in $F[x]$ with unknown coefficients $a_j \in F$ and known exponents $e_j \in \mathbb{N}$. For example suppose

$$a(x) = 3x^1 + 5x^4 + 7x^{11} + 9x^{13}.$$

To interpolate $a(x)$ of degree d, in general we need $d+1$ values of $a(x)$. But if $a(x)$ is sparse, like our example, we only need n values of $a(x)$ where n is the number of terms of $a(x)$, 4 in our example. For $v_1, v_2, \ldots, v_n \in F$, suppose we have computed $b_i = a(v_i)$ for $1 \le i \le n$ and we want to find the coefficients a_j of $a(x)$. Suppose also we use a geometric point sequence $v_i = \alpha^{i-1}$ for some $\alpha \in F$ such that $\alpha^i \ne \alpha^k$ for all $i \ne k$. Then

$$b_i = a(v_i) = \sum_{j=1}^{n} a_j (\alpha^{i-1})^{e_j} = \sum_{j=1}^{n} a_j (\alpha^{e_j})^{i-1} \text{ for } 1 \le i \le n.$$

If we let $u_j = \alpha^{e_j}$ for $1 \le j \le n$ then we have

$$b_i = \sum_{j=1}^{n} a_j u_j^{i-1}.$$

Authors' address: Hyukho Kwon, hyukhok@sfu.ca; Michael Monagan, mmonagan@sfu.ca, Department of Mathematics, Simon Fraser University, 8888 University Drive, Burnaby, British Columbia, Canada, V5A1A6.

In matrix-vector form, we have

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ u_1 & u_2 & u_3 & \cdots & u_n \\ u_1^2 & u_2^2 & u_3^2 & \cdots & u_n^2 \\ \vdots & \vdots & \vdots & & \vdots \\ u_1^{n-1} & u_2^{n-1} & u_3^{n-1} & \cdots & u_n^{n-1} \end{bmatrix}}_{U} \underbrace{\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}}_{\mathbf{b}}. \tag{1}$$

The matrix U is called a transposed Vandermonde matrix and the linear system $U\mathbf{a} = \mathbf{b}$ is called a transposed Vandermonde system.

Kaltofen and Yagati's algorithm [6] solves $U\mathbf{a} = \mathbf{b}$ using fast multiplication. If n is the dimension of U and $M(n)$ is the number of field operations for multiplying two polynomials of degree n in $F[x]$, Kaltofen and Yagati's algorithm does $O(M(n)\log n)$ arithmetic operations in F. Table 1 summarizes three methods for solving $U\mathbf{a} = \mathbf{b}$.

Methods	# ops in F	space
Gaussian Elimination	$O(n^3)$	$O(n^2)$
Zippel's method [9]	$O(n^2)$	$O(n)$
Kaltofen & Yagati's method [6]	$O(M(n)\log n)$	$O(n\log n)$

Table 1. Number of field operations for solving n by n transposed Vandermonde systems

We note that in some applications, for example, the GCD algorithm of Hu and Monagan in [5], one needs random evaluation points for v_i as $v_1 = \alpha^0 = 1$ may cause a problem. To resolve this we may instead use $v_i = \alpha^i$ for $1 \le i \le n$ so that $v_1 = \alpha$. This leads to a shifted transposed Vandermonde system $U'\mathbf{a} = \mathbf{b}$ where $U'_{i,j} = u_j^i$ for $1 \le i \le n$ and $1 \le j \le n$. The matrix U' factors as $U' = UD$ where D is a diagonal matrix with $D_{i,i} = u_i$. Thus to solve $U'\mathbf{a} = (UD)\mathbf{a} = \mathbf{b}$, since $\mathbf{a} = D^{-1}U^{-1}\mathbf{b}$. We first solve the unshifted transposed Vandermonde system $U\mathbf{c} = \mathbf{b}$ for $\mathbf{c}$ then use $\mathbf{a} = D^{-1}\mathbf{c}$.

2 Preliminaries

Kaltofen and Yagati's algorithm assumes fast multiplication, fast multi-point evaluation and fast division in $F[x]$. These algorithms are presented in [8] Chapter 8, 9 and 10. We summarize what we have implemented for these for the prime field $F = \mathbb{Z}_p$ where p is a 63 bit Fourier prime, a prime of the form $p = 2^k s + 1$ for large k. We will discuss the case when p is a non-Fourier prime in Section 4.

2.1 Fast multiplication

The underlying FFT for $\mathbb{Z}_p^n$ that we use is the in-place recursive FFT from Law and Monagan [7] which does exactly $\frac{1}{2}n\log_2 n$ multiplications. For ω a primitive nth root of unity in $\mathbb{Z}_p$, it precomputes the array W below of size n of the powers of ω needed for all recursive calls.

$$W = \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & \omega & \omega^2 & \cdots & \omega^{n/2-1} & 1 & \omega^2 & \omega^4 & \cdots & \omega^{n/2-2} & \cdots & 1 & 0 \\ \hline \end{array}$$

There are two types of FFTs. The decimation in frequency FFT permutes the elements at its last step. On the other hand, the decimation in time FFT permutes the elements in the beginning. The permutation in the FFT is called the bit-reversal permutation. To multiply two polynomials of degree at most n in $\mathbb{Z}_p[x]$, Law and Monagan use the decimation in frequency FFT for the two forward transforms and the decimation in time FFT for the inverse transform so that the two bit-reversal permutations cancel out and can be omitted from both FFTs.

THEOREM 2.1. *Fast multiplication does $M_1(n) = 9n\log_2 n + O(n)$ arithmetic operations to multiply two polynomials of degree at most n in $\mathbb{Z}_p[x]$.*

In our implementation, when the degree of the product of two polynomials is less than $2^{16} = 65536$, we use classical multiplication instead.

2.2 Fast division

Let $f, g \in \mathbb{Z}_p[x]$ with $\deg(g) = n$ and $\deg(f) < 2n$. The classical algorithm for f divided by g does $O(n^2)$ arithmetic operations in $\mathbb{Z}_p$. Let $g = \sum_{i=0}^{n} g_i x^i$ and $\widehat{g} = \sum_{i=0}^{n} g_{n-i}x^i$ denote the reciprocal polynomial. The fast division algorithm [8] computes $\widehat{g}^{-1} \bmod x^n$ using a Newton iteration (see Theorem 2.2 below) first.

THEOREM 2.2. *(Theorem 9.2 [8]) Assume $h = \sum_{i=0}^{n} h_i x^i \in \mathbb{Z}_p[x]$ and $h_0 \neq 0$. Let $y_0 = h_0^{-1}$ and $y_i = 2y_{i-1} - h \times y_{i-1}^2 \bmod x^{2^i}$ for $i \geq 1$. For all $i \geq 0$, $h \cdot y_i \bmod x^{2^i} = 1$.*

Using Theorem 2.2, we can compute the inverse of $\hat{g}$ in $3M_1(n) + O(n)$ arithmetic operations in $\mathbb{Z}_p$ [8]. We have implemented the middle product of Hanrot, Quercia, and Zimmermann [4] by rewriting the equation in Theorem 2.2 such that

$$y_i = 2y_{i-1} - h \times y_{i-1}^2 \bmod x^{2^i} = y_{i-1} + y_{i-1} \times (1 - h \times y_{i-1}) \bmod x^{2^i}. \tag{2}$$

Consider when $i = k$ and let $n = 2^k$. It follows that $h \times y_{k-1} \bmod x^{\frac{n}{2}} = 1$. Then we have

$$\begin{aligned}(h \bmod x^n) \times y_{k-1} \bmod x^n &= 1 + m_0 x^{\frac{n}{2}} + \cdots + m_{\frac{n}{2}-1}x^{n-1} + a_0 x^n + \cdots + a_{\frac{n}{2}-2}x^{\frac{3n}{2}-2} \bmod x^n \\ &= 1 + m \times x^{\frac{n}{2}} + a \times x^n \bmod x^n = 1 + m \times x^{\frac{n}{2}}\end{aligned}$$

where $m = \sum_{i=0}^{\frac{n}{2}-1} m_i x^i$ and $a = \sum_{i=0}^{\frac{n}{2}-2} a_i x^i$. The polynomial m is called the middle product. The main improvement is that we can use an FFT of size n instead of size $2n$ to compute $h \times y_{k-1} \bmod x^n$, and we can easily read off the coefficients of m and then

$$y_k = y_{k-1} + y_{k-1} \times (-m \times x^{\frac{n}{2}}) \bmod x^n.$$

Thus Hanrot, Quercia and Zimmermann's method reduces the cost of computing $\widehat{g}^{-1}$ from $3M_1(n) + O(n)$ to $2M_1(n) + O(n)$ arithmetic operations in $\mathbb{Z}_p$. Notice that there are two multiplications by y_{i-1} in Equation (2). We can also save one FFT by computing the FFT of y_{k-1} once which reduces the constant 2 to $\frac{5}{3}$. After computing $\widehat{g}^{-1}$, fast division computes the quotient q from $\widehat{q} = \widehat{f} \times \widehat{g}^{-1} \bmod x^n$ then the remainder r using $r = f - g \times q$. The total cost for the polynomial multiplications in division becomes $\frac{11}{3}M_1(n) + O(n)$ arithmetic operations in $\mathbb{Z}_p$.

THEOREM 2.3. *Fast division in $\mathbb{Z}_p[x]$ does at most $\frac{11}{3}M_1(n) + O(n)$ arithmetic operations.*

In our implementation of fast division we use classical division for $n \leq 512$.

2.3 Fast multi-point evaluation

Let f be a polynomial in $\mathbb{Z}_p[x]$ with $\deg(f) \leq n-1$ where $n = 2^k$ for some $k \geq 0$. Let $u_0, u_1, \ldots, u_{n-1}$ be distinct elements in $\mathbb{Z}_p$. The multi-point evaluation problem is to compute $f(u_i)$ for $0 \leq i \leq n-1$. Repeated usage of Horner's method costs $O(n^2)$ arithmetic operations in $\mathbb{Z}_p$.

In 1971, Borodin and Munro [2] introduced an $O(M_1(n)\log n)$ algorithm. Their algorithm first builds a product tree T, a complete binary tree in which every leaf is a linear polynomial $x - u_i$ for $0 \leq i \leq n-1$ and each parent node is the product of their two children so that the root node of T is $T_{k,0} = \prod_{i=0}^{n-1}(x - u_i)$. In Figure 1, we present the layout of a product tree.

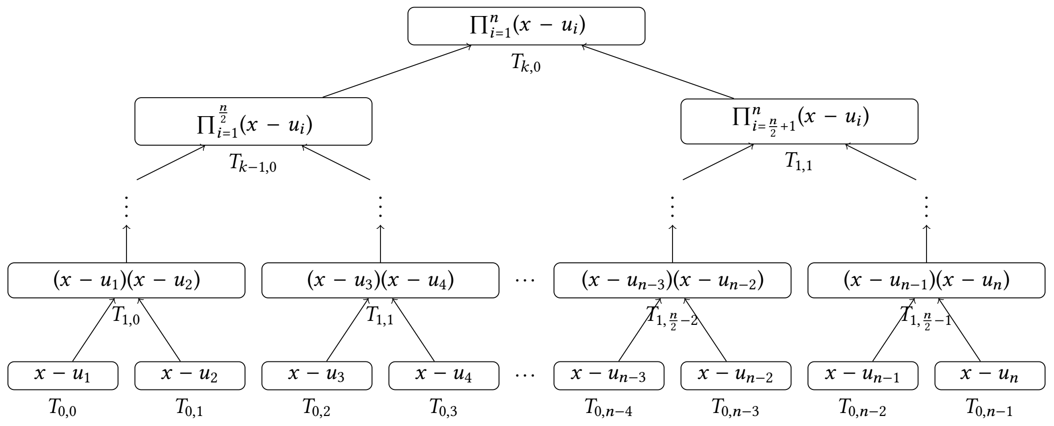

Fig. 1. The layout of a product tree

Each multiplication in the product tree is of two monic polynomials $T_{i,j} = x^{2^i} + A(x)$ by $T_{i,j+1} = x^{2^i} + B(x)$ which needs an FFT of size 4×2^i. Instead, by computing $T_{i,j} \times B(x)$ and adding $x^{2^i} T_{i,j}$ we can use an FFT of size 2×2^i. Also, since all polynomials in T are monic we only need to store $A(x)$ and $B(x)$ which have size 2^i. The product tree can be stored in a one dimensional array using space for $n(1 + \log_2 n)$ elements of $\mathbb{Z}_p$.

Theorem 2.4. *Building a product tree (BuPT) in $\mathbb{Z}_p[x]$ does $\frac{1}{2}M_1(n)\log_2 n + O(n \log n)$ arithmetic operations* [2].

We note that a product tree depends on distinct evaluation points, not a polynomial to be evaluated. Thus once we construct a product tree, we can use it to evaluate other polynomials at the same points.
Now let $m_i = x - u_i$ for all $0 \le i \le n-1$. The remainder of $f(x)$ divided m_i, denoted $f \bmod m_i$ is $f(u_i)$. Recall that if $g|h$, then $f \bmod g = (f \bmod h) \bmod g$ where $f, g, h \in \mathbb{Z}_p[x]$. Each node $T_{i,j}$ for $0 \le i \le k, 0 \le j < 2^{k-i}$ in the product tree T is a factor of its parent node in T. In other words,

$$T_{i,j} = T_{i-1,2j} \times T_{i-1,2j+1} \implies T_{i-1,2j}|T_{i,j} \text{ and } T_{i-1,2j+1}|T_{i,j}.$$

Thus we can compute $f(u_i)$ for all $0 \le i \le n-1$ by dividing down the product tree with a divide-and-conquer approach.

Theorem 2.5. *Dividing down the product tree (DDPT) in $\mathbb{Z}_p[x]$ does $\frac{11}{3}M_1(n)\log_2 n + O(n \log n)$ arithmetic operations.*

In the product tree, if $\deg T_{i,j} \le 64$ we use Horner's method because it is faster. Then we the space for T for $n \ge 64$ is $n(\log_2 n - 4)$.

3 Fast transposed Vandermonde solver

Kaltofen and Yagati's algorithm can be derived from Zippel's algorithm in [9].

3.1 Zippel's algorithm

From Equation (1), a transposed Vandermonde matrix U must have a unique inverse since we assume every u_i is distinct. Let $U^{-1} = (s_{i,j})_{1 \le i,j \le n}$. Then we can solve $U\mathbf{a} = \mathbf{b}$ by solving $\mathbf{a} = U^{-1}\mathbf{b}$. To compute U^{-1}, Zippel [9] sets $p_i \in \mathbb{Z}_p[x]$ to be the polynomial $p_i(x) = s_{i,1} + s_{i,2}x + \cdots + s_{i,n}x^{n-1}$

for $1 \le i \le n$. From the fact that $U^{-1}U = I$ where I is an $n \times n$ identity matrix, each entry $I_{i,j}$ is computed by multiplying the i-th row of U^{-1} by the j-th column of U. Observe that

$$I_{i,j} = \begin{bmatrix} s_{i,1} & s_{i,2} & \cdots & s_{i,n} \end{bmatrix} \begin{bmatrix} 1 \\ u_j \\ \vdots \\ u_j^{n-1} \end{bmatrix} = s_{i,1} \times 1 + s_{i,2} \times u_j + \cdots + s_{i,n} \times u_j^{n-1} = p_i(u_j).$$

Since I is the $n \times n$ identity matrix,

$$p_i(u_j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

To obtain the polynomials $p_i(x)$, let $M(x) \in \mathbb{Z}_p[x]$ be a polynomial such that $M(x) = \prod_{i=1}^{n}(x - u_i)$. Zippel calls $M(x)$ the master polynomial [9]. Let $q_i(x) \in \mathbb{Z}_p[x]$ of degree $n-1$ be computed by $q_i(x) = M(x)/(x - u_i) = \prod_{j \ne i}(x - u_j)$ for all $1 \le i \le n$. Then we have

$$\begin{cases} q_i(u_j) = 0 & \text{if } i \ne j \\ q_i(u_j) \ne 0 & \text{otherwise} \end{cases}$$

Then the coefficients of $p_i(x)$ can be obtained by computing $p_i(x) = q_i(u_i)^{-1} \times q_i(x)$ for $1 \le i \le n$ to satisfy the property in Equation (3). Consequently, we can obtain every entry of U^{-1}. Algorithm 1 shows Zippel's algorithm from [9]. The cost of the main steps is shown in the right column in Algorithm 1.

Algorithm 1 Zippel's transposed Vandermonde solver

```
Input: $n$, $\mathbf{u} \in \mathbb{Z}_p^n$ which defines the transposed Vandermonde matrix $U$, and $\mathbf{b} \in \mathbb{Z}_p^n$
Output: $\mathbf{a} \in \mathbb{Z}_p^n$ satisfying $U\mathbf{a} = \mathbf{b}$
 1: $M \longleftarrow x - u_1$
 2: for $i$ from 1 to $n-1$ do $M \longleftarrow M \times (x - u_{i+1})$ end for   // $M = \prod_{i=1}^{n}(x - u_i)$ ........... $O(n^2)$
 3: for $i$ from 1 to $n$ do
 4:     $q \longleftarrow M/(x - u_i)$ ........................................ $O(n)$
 5:     $c \longleftarrow q(u_i)$ .............................................. $O(n)$
 6:     if $c = 0$ then return ERROR "$u_i$'s are not distinct" end if
 7:     $r \longleftarrow c^{-1} \times q$     // Let $r = \sum_{j=0}^{n-1} r_j x^j$ .................... $O(n)$
 8:     $\mathbf{r} \longleftarrow [r_0, r_1, \ldots, r_{n-1}]$
 9:     $a_i \longleftarrow$ Compute the dot product of $\mathbf{r} \cdot \mathbf{b}$ ........................ $O(n)$
10: end for
11: return $[a_1, a_2, \ldots, a_n]$
```

Theorem 3.1. *Algorithm 1 does $O(n^2)$ arithmetic operations in $\mathbb{Z}_p$* [9].

3.2 Kaltofen and Yagati's algorithm

From Zippel's algorithm, note that

$$p_i(x) = q_i(u_i)^{-1} \times q_i(x) = s_{i,1} + s_{i,2}x + \cdots + s_{i,n}x^{n-1}.$$

Since $\mathbf{a} = U^{-1}\mathbf{b}$, $a_i = \sum_{j=1}^{n} s_{i,j} b_j$ for $1 \le i \le n$. Kaltofen and Yagati define the polynomial $D(x) \in \mathbb{Z}_p[x]$ such that

$$D(x) = b_n x + b_{n-1}x^2 + \cdots + b_1 x^n.$$

The coefficient of x^n in $p_i(x) \times D(x)$ is $\sum_{j=1}^{n} s_{i,j} b_j = a_i$ for $1 \le i \le n$. Then

$$p_i(x) = q_i(u_i)^{-1} \times q_i(x) = q_i(u_i)^{-1} \times \frac{M(x)}{(x-u_i)} = \frac{M(x)}{q_i(u_i) \times (x-u_i)}.$$

This implies that $M(x) = q_i(u_i) \times (x-u_i) \times p_i(x)$. Kaltofen and Yagati define the polynomial $H \in \mathbb{Z}_p[x]$ such that

$$H(x) = M(x) \times D(x) = h_0 x + h_1 x^2 + \cdots + h_{2n-2} x^{2n-1} + h_{2n-1} x^{2n}.$$

It follows that $H(x)/(x-u_i) = q_i(u_i) \times p_i(x) \times D(x)$. Thus the coefficient of x^n in $H(x)/(x-u_i)$ is $q_i(u_i) \times a_i$. In general, when we compute $H(x)/(x-z)$, the coefficient of x^i in the quotient is of the form $c_i(z) = h_i + h_{i+1} z + \cdots + h_{2n-1} z^{2n-1-i}$ and hence the coefficient of x^n in this quotient is

$$v(z) = c_n(z) = h_n + h_{n+1} z + \cdots + h_{2n-1} z^{n-1}.$$

Hence $v(u_i) = q_i(u_i) \times a_i$ for $1 \le i \le n$.
Recall that $q_i(x) = M(x)/(x-u_i)$. The derivative of $M(x)$ can be expressed as

$$M'(x) = (x-u_i) \times q_i'(x) + (x-u_i)' \times q_i(x) = (x-u_i) \times q_i'(x) + q_i(x).$$

By evaluating M' at u_i we have

$$M'(u_i) = (u_i - u_i) \times q_i'(u_i) + q_i(u_i) = q_i(u_i).$$

Hence $a_i = v(u_i)/q_i(u_i) = v(u_i)/M'(u_i)$ for $1 \le i \le n$. We present Kaltofen and Yagati's algorithm from [6] in Algorithm 2. From $a_i = v(u_i)/M'(u_i)$, we use the product tree to obtain $M(x) = \prod_{i=1}^{n}(x-u_i)$ in step 2 and to compute $v(u_i)$ in step 6 and $M'(u_i)$ in step 8 in Algorithm 2.

Algorithm 2 Kaltofen and Yagati's fast transposed Vandermonde solver (FastTVS)

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $\mathbf{u} \in \mathbb{Z}_p^n$ which defines the transposed Vandermonde matrix $U$, and
    $\mathbf{b} \in \mathbb{Z}_p^n$
Output: $\mathbf{a} \in \mathbb{Z}_p^n$ satisfying $U\mathbf{a} = \mathbf{b}$.
 1: $T \leftarrow$ BuPT$(n, u)$ ........ $\frac{1}{2}M_1(n)\log_2 n + O(n \log n)$
 2: $M \leftarrow T_{k,0}$ from $T$   // $M = \prod_{i=1}^{n}(x-u_i)$
 3: $D \leftarrow b_n x + b_{n-1}x^2 + \cdots + b_1 x^n$
 4: $H \leftarrow M \times D$   // Let $H = \sum_{i=0}^{2n-1} h_i x^{i+1}$ ........ $M_1(n)$
 5: $v \leftarrow \sum_{i=0}^{n-1} h_{n+i} x^i$   // $\sum_{i=0}^{n-1} h_{n+i} z^i$ is the coefficient of $x^n$ in $H/(x-z)$
 6: $s_1, s_2, \ldots, s_n \leftarrow$ DDPT$(n, v, T)$   // $s_i = v(u_i)$ ........ $\frac{11}{3}M_1(n)\log_2 n + O(n \log n)$
 7: Differentiate $M$ ........ $O(n)$
 8: $t_1, t_2, \ldots, t_n \leftarrow$ DDPT$(n, M', T)$   // $t_i = M'(u_i) = q_i(u_i)$ ........ $\frac{11}{3}M_1(n)\log_2 n + O(n \log n)$
 9: for $i$ from 1 to $n$ do $a_i \leftarrow t_i^{-1} \cdot s_i$ end for ........ $O(n)$
10: return $[a_1, a_2, \ldots, a_n]$
```

Theorem 3.2. *Algorithm 2 (FastTVS) does at most $\frac{53}{6}M_1(n)\log_2 n + O(n \log n)$ arithmetic operations in $\mathbb{Z}_p$.*

In our implementation of Algorithm 2, for $n \le 64$ we evaluate using Horner's method instead of dividing down the product tree. Also, if after Step 1 we compute the inverses of all $\widehat{T_{i,j}}$ polynomials, we can use them for both DDPT calls in Steps 6 and 8. Computing intermediate inverses costs $\frac{5}{3}M_1(n)\log_2 n + O(n \log n)$. This reduces the cost of each DDPT to $2M_1(n)\log_2 n + O(n \log n)$ arithmetic operations. Hence the total cost of Algorithm 2 is reduced to $\frac{43}{6}M_1(n)\log_2 n + O(n \log n)$. This optimization reduced the time for $n = 2^{16}$ in Table 2 from 1500.7 ms to 1249.3 ms.

4 Fast transposed Vandermonde solver over a non-Fourier prime field

Our presented algorithm and analysis of Kaltofen and Yagati's algorithm assumes that p is a Fourier prime. However, when p is not a Fourier prime, we cannot do the polynomial multiplication that arises in the algorithm using the FFT. In this case, we can apply the three primes method from [8], which essentially multiplies the cost of fast multiplication, fast division, fast multi-point evaluation and Kaltofen and Yagati's algorithm by a factor of three.

Let $a(x) = \sum_{i=1}^{t} a_i x^{e_i}$ where the coefficients are integers. Define $||a||_\infty = \max_{i=1}^{t} |a_i|$ the height of $a(x)$. Let $f, g \in \mathbb{Z}_p[x]$ where p is not a Fourier prime. Treating the coefficients of f and g as integers, since $||f||_\infty < p$ and $||g||_\infty < p$, $||f \times g||_\infty < (p-1)^2(1+\min(\deg(f), \deg(f)))$. To multiply modulo p we choose three distinct Fourier primes p_1, p_2, and p_3 such that $p_1p_2p_3 > (p-1)^2(1+\min(\deg(f), \deg(g)))$. We use the FFT to multiply $f \times g \mod p_i$ for $i = 1, 2, 3$ and then Chinese remaindering to recover the integer coefficients in $f \times g$ before reduction mod p. In [8], von zur Gathen and Gerhard use it for multiplying long integers. Fast multiplication with the three primes method is presented in Algorithm 3.

Algorithm 3 Fast multiplication with three primes (FastMul3p)

```
Input: Polynomials $f, g \in \mathbb{Z}_p[x]$ such that $f = \sum_{i=0}^{d_1} f_i x^i$ and $g = \sum_{i=0}^{d_2} g_i x^i$
Output: $h = f \times g \in \mathbb{Z}_p[x]$
 1: Pick three distinct Fourier primes $p_1, p_2, p_3$ such that $p_1p_2p_3 > (p-1)^2(1+\min(\deg(f), \deg(g)))$
 2: $a \leftarrow$ Compute $(f \bmod p_1) \times (g \bmod p_1)$ using fast multiplication // Let $a = \sum_{i=0}^{d_1+d_2} a_i x^i$
 3: $b \leftarrow$ Compute $(f \bmod p_2) \times (g \bmod p_2)$ using fast multiplication // Let $b = \sum_{i=0}^{d_1+d_2} b_i x^i$
 4: $c \leftarrow$ Compute $(f \bmod p_3) \times (g \bmod p_3)$ using fast multiplication // Let $c = \sum_{i=0}^{d_1+d_2} c_i x^i$
 5: for $i$ from 0 to $d_1 + d_2$ do
 6:     $h_i \leftarrow$ Solve $\{h_i \equiv a_i \bmod p_1, h_i \equiv b_i \bmod p_2, h_i \equiv c_i \bmod p_3\}$ for $0 \le h_i < p_1p_2p_3$
 7:     $h_i \leftarrow h_i \bmod p$
 8: end for
 9: $h \leftarrow \sum_{i=0}^{d_1+d_2} h_i x^i$
10: return $h$
```

Theorem 4.1. *Algorithm 3 does $M_2(n) = 3M_1(n) = 3(9n\log_2 n + O(n)) = 27n\log_2 n + O(n)$ field operations to multiply two polynomials of degree at most n in $\mathbb{Z}_p[x]$.*

Thus if p is not a Fourier prime with $p = 2^k s + 1$ with $2^k > d_1 + d_2$, the three primes method is used in every polynomial multiplication in the subroutines in the fast transposed Vandermonde solver.

Theorem 4.2. *The fast transposed Vandermonde solver with the three primes method does at most $\frac{43}{6}M_2(n)\log_2 n + O(n\log n) = \frac{43}{2}M_1(n)\log_2 n + O(n\log n)$ field operations.*

5 Implementation and Benchmark

We have implemented Kaltofen and Yagati's algorithm (Algorithm 2 FastTVS) in C for the case where p is a 63 bit Fourier prime.
For comparison we have also implemented Zippel's $O(n^2)$ algorithm from [9] in C and we have also implemented Kaltofen and Yagati's algorithm in Maple. The Maple implementation for multiplication in $\mathbb{Z}_p[x]$ is done using a single large integer multiplication using GMP's fast integer multiplication.

We have also implemented Algorithm 2 with the three primes method where p is a 57 bit non Fourier prime. To use the three primes method, we have used the three Fourier primes: a 32 bit prime $p_1 = 3 \cdot 2^{30} + 1$, a 62 bit prime $p_2 = 69 \cdot 2^{55} + 1$, and another 62 bit prime $p_3 = 29 \cdot 2^{57} + 1$. The timings in Table 2 and Table 3 are in milliseconds. They were obtained on an AMD FX 8350-8 8-core CPU at 4.2GHz using one core.

Table 2. CPU timings in *ms* for solving $n \times n$ transposed Vandermonde system over $\mathbb{Z}_p$ with $p = 116 \cdot 2^{55} + 1$

n	FastTVS					Zippel	Zippel /	Maple	Maple /
	BuPT	InvTree	DDPT1	DDPT2	total	time	FastTVS	time	FastTVS
2^6	0.046	-	0.046	0.039	0.195	0.1389	0.71	3.4	13.7
2^7	0.086	-	0.107	0.098	0.380	0.4879	1.28	8.6	22.6
2^8	0.150	-	0.254	0.238	0.808	1.9039	2.35	20.8	25.7
2^9	0.363	-	0.693	0.674	2.065	7.4640	3.61	63.0	30.5
2^{10}	0.875	0.600	1.890	1.877	5.811	30.826	5.30	113.2	19.5
2^{11}	2.020	2.417	5.070	5.008	15.775	116.84	7.40	270.0	17.1
2^{12}	4.755	7.529	12.307	12.268	39.444	469.64	11.90	608.0	15.3
2^{13}	11.146	20.556	29.566	29.270	95.765	1,868.0	19.50	1,321	13.8
2^{14}	25.901	53.099	71.091	70.580	231.55	7,455.7	32.19	3,025	13.1
2^{15}	60.151	131.30	166.15	166.46	546.52	29,986	54.86	7,190	13.1
2^{16}	131.23	314.56	380.02	376.77	1,249.3	120,292	96.28	16,455	13.2
2^{17}	339.89	746.70	867.30	863.48	2,914.9	478,912	164.3	69,705	23.9
2^{18}	663.01	1,747.1	1,961.8	1,955.2	6,529.8	1,929,776	295.5	97,667	14.9

Table 3. CPU timings in *ms* for solving $n \times n$ transposed Vandermonde system over $\mathbb{Z}_p$ with $p = 144115188075855859 < 2^{57}$ using the three primes method

n	FastTVS					Zippel	Zippel /
	BuPT	InvTree	DDPT1	DDPT2	total		FastTVS
2^6	0.043	-	0.042	0.040	0.247	0.1509	0.61
2^7	0.059	-	0.145	0.144	0.545	0.4869	0.89
2^8	0.276	-	0.343	0.340	1.355	1.9060	1.40
2^9	0.899	-	0.857	0.843	3.463	7.4809	2.16
2^{10}	2.615	-	2.388	2.398	9.195	29.702	3.23
2^{11}	6.685	-	7.510	7.374	25.353	116.44	4.59
2^{12}	16.044	-	25.265	25.139	73.946	470.04	6.35
2^{13}	38.754	80.014	76.846	76.688	288.00	1865.1	6.47
2^{14}	93.628	212.80	213.94	214.83	768.44	7478.1	9.73
2^{15}	214.23	510.61	541.49	540.96	1,875.7	29,763	15.86
2^{16}	497.72	1,237.4	1,343.2	1,354.4	4,576.6	119,478	26.10
2^{17}	1,111.9	2,890.1	3,199.1	3,210.3	10,716	488,369	45.57
2^{18}	2,494.2	6,632.9	7,480.6	7,470.6	24,725	1,953,252	78.99

In Table 2 column BuPT is the time for building the product tree, column InvTree is the time to construct the inverse product tree, columns DDPT1 and DDPT2 are the time for dividing down the product tree using the inverse product tree, and column total is total time for Algorithm 2. Column Zippel is the time for Zippel's $O(n^2)$ algorithm in C and column Maple is the time for our

Maple implementation of Kaltofen and Yagati's algorithm. Column Zippel/FastTVS is the speedup of our FastTVS compared with Zippel and column Maple/FastTVS is the slowdown of the Maple implementation of Kaltofen and Yagati.

According to Table 2, our FastTVS implementation beats Zippel's $O(n^2)$ method for $n \geq 128$ which is a good result. Also, the Maple time of 69,705 ms for $n = 2^{17}$ is not an error; it seems to be an anomaly.

From Table 3, our FastTVS with the three prime method beats Zippel's quadratic method for $n \geq 256$ as well. Notice also that the time to build the product tree (BuPT) is much smaller than the time to compute the inverses (column InvTree) plus divide down the product tree (columns DDPT1 and DDPT2) in both tables. Thus any improvement will need to focus on polynomial division.

FastTVS, our C implementation of Kaltofen and Yagati, is 13 to 30 times faster than our pure Maple implementation in Table 2. One reason for this is for $p = 116 \cdot 2^{55} + 1$, which is a Fourier prime, we do not need the three primes method so we gain a factor of 3 over the Maple implementation. Also, we compute the inverse product tree once which saves a further 20% of the work.

References

[1] Michael Ben-Or and Prasoon Tiwari: A Deterministic Algorithm for Sparse Multivariate Polynomial Interpolation. *Proceedings of STOC '20*, pp. 301–309, ACM, 1988.

[2] A. Borodin and I. Munro: Evaluating polynomials at many points. *Information Processing Letters **1(2):*** pp. 66–68, 1971.

[3] Tian Chen and Michael Monagan: Factoring Multivariate Polynomials Represented by Black Boxes – A Maple + C Implementation. *Mathematics in Computer Science **16***(2–3), article 18, Springer, 2022. https://doi.org/10.1007/s11786-022-00534-7

[4] Guillaume Hanrot, Michel Quercia, Paul Zimmermann: The Middle Product Algorithm I. Speeding up the division and square root of power series. *Applicable Algebra in Engineering, Communication and Computing **14(6):*** pp. 415–438, Springer, 2004.

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

[6] Erich Kaltofen and Lakshman Yagati: Improved sparse multivariate polynomial interpolation algorithms. *Proceedings of ISSAC'88,* pp. 467–474, Springer, 1988.

[7] Marshall Law and Michael Monagan: A parallel implementation for polynomial multiplication modulo a prime. *Proceedings of PASCO'2015,* pp. 78–86, ACM, 2015.

[8] Joachim von zur Gathen and Jüergen Gerhard: *Modern Computer Algebra,* Cambridge University Press, 2013.

[9] Richard Zippel: Interpolating polynomials from their values. *Journal of Symbolic Computation **9(3):*** pp. 375–403, Elsevier Ltd, 1990.

Acknowledgments

This work was supported by NSERC of Canada and Maplesoft.