On the Complexity and Parallel Implementation of Hensel's Lemma and Weierstrass Preparation

Alexander Brandt, Marc Moreno Maza

Department of Computer Science, The University of Western Ontario,
London, Canada
`abrandt5@uwo.ca`, `moreno@csd.uwo.ca`

Abstract

Hensel's lemma, combined with repeated applications of Weierstrass preparation theorem, allows for the factorization of polynomials with multivariate power series coefficients. We present a complexity analysis for this method and leverage those results to guide the load-balancing of a parallel implementation to concurrently update all factors. In particular, the factorization creates a *pipeline* where the terms of degree k of the first factor are computed simultaneously with the terms of degree $k-1$ of the second factor, etc. An implementation challenge is the inherent irregularity of computational work between factors, as our complexity analysis reveals. Additional resource utilization and load-balancing is achieved through the parallelization of Weierstrass preparation. Experimental results show the efficacy of this mixed parallel scheme, achieving up to 9× parallel speedup on a 12-core machine.

Keywords: Formal power series · Weierstrass preparation · Hensel's lemma · Hensel factorization · Parallel processing · Parallel pipeline

1 Introduction

Factorization via Hensel's lemma, or simply Hensel factorization, provides a mechanism for factorizing univariate polynomials with multivariate power series coefficients. In particular, for a multivariate polynomial in $(X_1, \ldots, X_n, Y)$, monic and square-free as a polynomial in Y, one can compute its roots with respect to Y as power series in $(X_1, \ldots, X_n)$. For a bivariate polynomial in (X_1, Y), the classical Newton–Puiseux method is known to compute the polynomial's roots with respect to Y as univariate Puiseux series in X_1. The transition from power series to Puiseux series arises from handling the non-monic case.

The *Hensel–Sasaki Construction* or *Extended Hensel Construction* (EHC) was proposed in [21] as an efficient alternative to the Newton–Puiseux method for the case of univariate coefficients. In the same paper, an extension of the Hensel–Sasaki construction for multivariate coefficients was proposed, and then later extended, see e.g., [14, 22]. In [1], EHC was improved in terms of algebraic complexity and practical implementation.

In this paper, we present a parallel algorithm and its implementation for Hensel factorization based on repeated applications of Weierstrass preparation theorem. Our method uses a *lazy evaluation* scheme, meaning that more terms can be computed on demand without having to restart the computation. This contrasts with a *truncated* implementation where only terms up to a pre-determined degree are computed. Unfortunately, such a degree often cannot be determined before calculations start, or later may be found to not go far enough. This scenario occurs, for instance, when computing limits of real rational functions [1].

Lazy evaluation is not new, having previously been employed in sparse polynomial arithmetic [19] and *univariate* power series arithmetic [9, 26]. Our previous work in [8] is, to the best of our knowledge, the first lazy multivariate power series implementation. Our implementation of lazy and parallel power series supports an arbitrary number of variables. However, the complexity estimates of our proposed methods are measured in the bivariate case; see Section 4. This allows us to obtain sharp complexity estimates, giving the number of operations required to update each factor of a Hensel factorization individually. This information helps guide and load-balance our parallel implementation. Further, limiting to the bivariate case allows for comparison with existing works.

Denote by $M(n)$ a polynomial multiplication time [28, Ch. 8] (the cost sufficient to multiply two polynomials of degree n), Let $\mathbb{K}$ be algebraically closed and $f \in \mathbb{K}[[X_1]][Y]$ have degree d_Y in Y and total degree d. Our Hensel factorization computes the first k terms of all factors of f within $\mathcal{O}(d_Y^3 k + d_Y^2 k^2)$ operations in $\mathbb{K}$. We conjecture in Section 4 that we can achieve $\mathcal{O}(d_Y^3 k + d_Y^2 M(k) \log k)$ using relaxed algorithms [26]. The EHC of [1] computes the first k terms of all factors in $\mathcal{O}(d^3 M(d) + k^2 d M(d))$. Kung and Traub show that, over the complex numbers $\mathbb{C}$, the Newton–Puiseux method can do the same in $\mathcal{O}(d^2 k M(k))$ (resp. $\mathcal{O}(d^2 M(k))$) operations in $\mathbb{C}$ using a linear lifting (resp. quadratic lifting) scheme [15]. This complexity is lowered to $\mathcal{O}(d^2 k)$ by Chudnovsky and Chudnovsky in [10]. Berthomieu, Lecerf, and Quintin in [7] also present an algorithm and implementation based on Hensel lifting which performs in $\mathcal{O}(M(d_Y) \log(d_Y)\, k M(k))$; this is better than previous methods with respect to d (or d_Y), but worse with respect to k.

However, these estimates ignore an initial root finding step. Denote by $R(n)$ the cost of finding the roots in $\mathbb{K}$ of a degree n polynomial (e.g. [28, Th. 14.18]). Our method then performs in $\mathcal{O}(d_Y^3 k + d_Y^2 k^2 + R(d_Y))$. Note that the $R(d_Y)$ term does not depend on k, and is thus ignored henceforth. For comparison, however, Neiger, Rosenkilde, and Schost in [20] present an algorithm based on Hensel lifting which, *ignoring polylogarithmic factors*, performs in $\mathcal{O}(d_Y k + k R(d_Y))$.

Nonetheless, despite a higher asymptotic complexity, the formulation of EHC in [1] is shown to be practically much more efficient than that of Kung and Traub. Our serial implementation of lazy Hensel factorization (using plain, quadratic arithmetic) has already been shown in [8] to be orders of magnitude faster than that implementation of EHC. Similarly, in [8], we show that our serial lazy power series is orders of magnitude faster than the truncated implementations of MAPLE's [16] `mtaylor` and SAGEMATH's [25] `PowerSeriesRing`. This highlights that a lazy scheme using suboptimal routines—but a careful implementation—can still be practically efficient despite higher asymptotic complexity.

Further still, it is often the case that asymptotically fast algorithms are much more difficult to parallelize, and have high parallel overheads, e.g. polynomial multiplication based on FFT. Hence, in this work, we look to improve the practical performance (i.e. when $k \gg d$) of our previous lazy implementation through the use of parallel processing rather than by reducing asymptotic bounds.

In Hensel factorization, computing power series terms of each factor relies on the computed terms of the previous factor. In particular, the output of one Weierstrass preparation becomes the input to another. These successive dependencies naturally lead to a parallel *pipeline* or chain of *producer-consumer* pairs. Within numerical linear algebra, pipelines have already been employed in parallel implementations of singular value decomposition [13], LU decomposition, and Gaussian elimination [18]. Meanwhile, to the best of our knowledge, the only use of parallel pipeline in symbolic computation is [5], which examines a parallel implementation of triangular decomposition of polynomial systems.

However, in our case, work reduces with each pipeline stage, limiting throughput. To overcome this challenge, we first make use of our complexity estimates to dynamically estimate the work required to update each factor. Second, we compose parallel schemes by applying the celebrated map-reduce pattern within Weierstrass preparation, and thus within a stage of the pipeline. Assigning multiple threads to a single pipeline stage improves load-balance and increases throughput. Experimental results show this composition is effective, with a parallel speedup of up to 9× on a 12-core machine.

The remainder of this paper is organized as follows. Section 2 reviews mathematical background and notations. Further background on our lazy power series of [8] is presented in Section 3. Algorithms and complexity analyses of Weierstrass preparation and Hensel factorization are given in Section 4. Section 5 presents our parallel variations, where our complexity estimates are used for dynamic scheduling. Finally, Section 6 discusses experimental data.

2 Background

We take this section to present basic concepts and notation of multivariate power series and univariate polynomials over power series (UPoPS). Further, we present constructive proofs for the theorems of Weierstrass preparation and Hensel's lemma for UPoPS, from which algorithms are adapted; see Sections 4.1 and 4.2. Further introductory details may be found in the book of G. Fischer [11].

2.1 Power Series and Univariate Polynomials over Power Series

Let $\mathbb{K}$ be an algebraically closed field. We denote by $\mathbb{K}[[X_1, \ldots, X_n]]$ the ring of formal power series with coefficients in $\mathbb{K}$ and with variables $X_1, \ldots, X_n$.

Let $f = \sum_{e \in \mathbb{N}^n} a_e X^e$ be a formal power series, where $a_e \in \mathbb{K}$, $X^e = X_1^{e_1} \cdots X_n^{e_n}$, $e = (e_1, \ldots, e_n) \in \mathbb{N}^n$, and $|e| = e_1 + \cdots + e_n$. Let k be a non-negative integer. The *homogeneous part* of f in degree k, denoted $f_{(k)}$, is defined by $f_{(k)} = \sum_{|e|=k} a_e X^e$. The *order* of f, denoted $\mathrm{ord}(f)$, is defined as $\min\{i \mid f_{(i)} \neq 0\}$, if $f \neq 0$, and as ∞ otherwise.

Recall several properties regarding power series. First, $\mathbb{K}[[X_1, \ldots, X_n]]$ is an integral domain. Second, the set $\mathcal{M} = \{f \in \mathbb{K}[[X_1, \ldots, X_n]] \mid \mathrm{ord}(f) \geq 1\}$ is the only maximal ideal of $\mathbb{K}[[X_1,\ldots,X_n]]$. Third, for all $k \in \mathbb{N}$, we have $\mathcal{M}^k = \{f \in \mathbb{K}[[X_1,\ldots,X_n]] \mid \operatorname{ord}(f) \geq k\}$. Note that for $n = 0$ we have $\mathcal{M} = \langle 0 \rangle$. Further, note that $f_{(k)} \in \mathcal{M}^k \setminus \mathcal{M}^{k+1}$ and $f_{(0)} \in \mathbb{K}$. Fourth, a unit $u \in \mathbb{K}[[X_1,\ldots,X_n]]$ has $\operatorname{ord}(u) = 0$ or, equivalently, $u \notin \mathcal{M}$.

Let $f, g, h, p \in \mathbb{K}[[X_1,\ldots,X_n]]$. The *sum* and *difference* $f = g \pm h$ is given by $\sum_{k\in\mathbb{N}} (g_{(k)} \pm h_{(k)})$. The product $p = g\,h$ is given by $\sum_{k\in\mathbb{N}} \left(\Sigma_{i+j=k}\, g_{(i)}h_{(j)}\right)$. Notice that the these formulas naturally suggest a *lazy evaluation* scheme, where the result of an arithmetic operation can be incrementally computed for increasing *precision*. A power series f is said to be known to precision $k \in \mathbb{N}$, when $f_{(i)}$ is known for all $0 \leq i \leq k$. Such an update function, parameterized by k, for addition or subtraction is simply $f_{(k)} = g_{(k)} \pm h_{(k)}$; an update function for multiplication is $p_{(k)} = \sum_{i=0}^{k} g_{(i)}h_{(k-i)}$. Lazy evaluation is discussed further in Section 3. From these update formulas, the following observation follows.

Observation 2.1 (power series arithmetic) *Let* $f, g, h, p \in \mathbb{K}[[X_1]]$ *with* $f = g \pm h$ *and* $p = g\,h$. $f_{(k)} = g_{(k)} \pm h_{(k)}$ *can be computed in 1 operation in* $\mathbb{K}$. $p_{(k)} = \sum_{i=0}^{k} g_{(i)}h_{(k-i)}$ *can be computed in* $2k-1$ *operations in* $\mathbb{K}$.

Now, let $f, g \in \mathbb{A}[Y]$ be univariate polynomials over power series where $\mathbb{A} = \mathbb{K}[[X_1,\ldots,X_n]]$. Writing $f = \sum_{i=0}^{d} a_i Y^i$, for $a_i \in \mathbb{A}$ and $a_d \neq 0$, we have that the degree of f (denoted $\deg(f, Y)$ or simply $\deg(f)$) is d. Note that arithmetic operations for UPoPS are easily derived from the arithmetic of its power series coefficients. A UPoPS is said to be known up to precision k if each of its power series coefficients are known up to precision k. A UPoPS f is said to be *general (in Y) of order j* if $f \bmod \mathcal{M}[Y]$ has order j when viewed as a power series in Y. Thus, for $f \notin \mathcal{M}[Y]$, writing $f = \sum_{i=0}^{d} a_i Y^i$, we have $a_i \in \mathcal{M}$ for $0 \leq i < j$ and $a_j \notin \mathcal{M}$.

2.2 Weierstrass Preparation Theorem and Hensel Factorization

The Weierstrass Preparation Theorem (WPT) is fundamentally a theorem regarding factorization. In the context of analytic functions, WPT implies that any analytic function resembles a polynomial in the neighbourhood of the origin. Generally, WPT can be stated for power series over power series, i.e. $\mathbb{A}[[Y]]$. This can be used to prove that $\mathbb{A}$ is both a unique factorization domain and a Noetherian ring. See [8] for such a proof of WPT. Here, it is sufficient to state the theorem for UPoPS. First, we begin with a simple lemma.

Lemma 2.2 *Let* $f, g, h \in \mathbb{K}[[X_1,\ldots,X_n]]$ *such that* $f = gh$. *Let* $f_i = f_{(i)}, g_i = g_{(i)}, h_i = h_{(i)}$. *If* $f_0 = 0$ *and* $h_0 \neq 0$, *then* g_k *is uniquely determined by* $f_1,\ldots,f_k$ *and* $h_0,\ldots,h_{k-1}$

Proof. We proceed by induction on k. Since $f_0 = g_0h_0 = 0$ and $h_0 \neq 0$ both hold, the statement holds for $k = 0$. Now let $k > 0$, assuming the hypothesis holds for $k-1$. To determine g_k it is sufficient to expand $f = gh$ modulo $\mathcal{M}^{k+1}$: $f_1 + f_2 + \cdots + f_k = g_1h_0 + (g_1h_1 + g_2h_0) + \cdots + (g_1h_{k-1} + \cdots + g_{k-1}h_1 + g_kh_0)$; and, recalling $h_0 \in \mathbb{K} \setminus \{0\}$, we have $g_k = {}^1\!/_{h_0}\,(f_k - g_1h_{k-1} - \cdots - g_{k-1}h_1)$. $\square$

Theorem 2.3 (Weierstrass Preparation Theorem) *Let f be a polynomial of $\mathbb{K}[[X_1,\ldots,X_n]][Y]$ so that $f \not\equiv 0 \mod \mathcal{M}[Y]$ holds. Write $f = \sum_{i=0}^{d+m} a_i Y^i$, with $a_i \in \mathbb{K}[[X_1,\ldots,X_n]]$, where $d \geq 0$ is the smallest integer such that $a_d \notin \mathcal{M}$ and m is a non-negative integer. Assume $f \not\equiv 0 \mod \mathcal{M}[Y]$. Then, there exists a unique pair p, α satisfying the following:*

(i) $f = p\,\alpha$,

(ii) *α is an invertible element of* $\mathbb{K}[[X_1,\ldots,X_n]][[Y]]$,

(iii) *p is a monic polynomial of degree d,*

(iv) *writing $p = Y^d + b_{d-1}Y^{d-1} + \cdots b_1 Y + b_0$, we have $b_{d-1},\ldots,b_0 \in \mathcal{M}$.*

Proof. If $n = 0$, writing $f = \alpha Y^d$ with $\alpha = \sum_{i=0}^{m} a_{i+d} Y^i$ proves the existence of the decomposition. Now, assume $n \geq 1$. Write $\alpha = \sum_{i=0}^{m} c_i Y^i$, with $c_i \in \mathbb{K}[[X_1,\ldots,X_n]]$. We will determine $b_0,\ldots,b_{d-1},c_0,\ldots,c_m$ modulo successive powers of $\mathcal{M}$. Since we require α to be a unit, $c_0 \notin \mathcal{M}$ by definition. $a_0,\ldots,a_{d-1}$ are all 0 mod $\mathcal{M}$. Then, equating coefficients in $f = p\,\alpha$ we have:

$$\begin{array}{rcl} a_0 & = & b_0 c_0 \\ a_1 & = & b_0 c_1 + b_1 c_0 \\ & \vdots & \\ a_{d-1} & = & b_0 c_{d-1} + b_1 c_{d-2} + \cdots + b_{d-2} c_1 + b_{d-1} c_0 \\ a_d & = & b_0 c_d + b_1 c_{d-1} + \cdots + b_{d-1} c_1 + c_0 \\ & \vdots & \\ a_{d+m-1} & = & b_{d-1} c_m + c_{m-1} \\ a_{d+m} & = & c_m \end{array} \tag{1}$$

and thus $b_0,\ldots,b_{d-1}$ are all 0 mod $\mathcal{M}$. Then, $c_i \equiv a_{d+i} \mod \mathcal{M}$ for all $0 \leq i \leq m$. All coefficients have thus been determined mod $\mathcal{M}$. Let $k \in \mathbb{Z}^+$. Assume inductively that all $b_0,\ldots,b_{d-1},c_0,\ldots,c_m$ have been determined mod $\mathcal{M}^k$.

It follows from Lemma 2.2 that b_0 can be determined mod $\mathcal{M}^{k+1}$ from the equation $a_0 = b_0 c_0$. Consider now the second equation. Since b_0 is known mod $\mathcal{M}^{k+1}$, and $b_0 \in \mathcal{M}$, the product $b_0 c_1$ is also known mod $\mathcal{M}^{k+1}$. Then, we can determine b_1 using Lemma 2.2 and the formula $a_1 - b_0 c_1 = b_1 c_0$. This procedure follows for $b_2,\ldots,b_{d-1}$. With $b_0,\ldots,b_{d-1}$ known mod $\mathcal{M}^{k+1}$ each $c_0,\ldots,c_m$ can be determined mod $\mathcal{M}^{k+1}$ from the last $m+1$ equations. □

One requirement of WPT is that $f \not\equiv 0 \mod \mathcal{M}[Y]$. That is to say, f cannot vanish at $(X_1,\ldots,X_n) = (0,\ldots,0)$ and, specifically, f is general of order $d = \deg(p)$. A suitable linear change in coordinates can always be applied to meet this requirement; see Algorithm 2 in Section 4. Since Weierstrass preparation provides a mechanism to factor a UPoPS into two factors, suitable changes in coordinates and several applications of WPT can fully factorize a UPoPS. The existence of such a factorization is given by Hensel's lemma for UPoPS.

Theorem 2.4 (Hensel's Lemma) *Let $f = Y^d + \sum_{i=0}^{d-1} a_i Y^i$ be a monic polynomial with $a_i \in \mathbb{K}[[X_1,\ldots,X_n]]$. Let $\bar{f} = f(0,\ldots,0,Y) = (Y - c_1)^{d_1}(Y - c_2)^{d_2}\cdots(Y - c_r)^{d_r}$ for $c_1,\ldots,c_r \in \mathbb{K}$ and positive integers $d_1,\ldots,d_r$. Then, there exists $f_1,\ldots,f_r \in \mathbb{K}[[X_1,\ldots,X_n]][Y]$, all monic in Y, such that:*

(i) $f = f_1 \cdots f_r$,

(ii) $\deg(f_i, Y) = d_i$ *for* $1 \le i \le r$*, and*

(iii) $\bar{f}_i = (Y - c_i)^{d_i}$ *for* $1 \le i \le r$.

Proof. We proceed by induction on r. For $r = 1$, $d_1 = d$ and we have $f_1 = f$, where f_1 has all the required properties. Now assume $r > 1$. A change of coordinates in Y, sends c_r to 0. Define $g(X_1, \ldots, X_n, Y) = f(X_1, \ldots, X_n, Y + c_r) = (Y + c_r)^d + a_{d-1}(Y + c_r)^{d-1} + \cdots + a_0$. By construction, g is general of order d_r and WPT can be applied to obtain $g = p\,\alpha$ with p being of degree d_r and $\bar{p} = Y^{d_r}$. Reversing the change of coordinates we set $f_r = p(Y - c_r)$ and $f^* = \alpha(Y - c_r)$, and we have $f = f^* f_r$. f_r is a monic polynomial of degree d_r in Y with $\bar{f}_r = (Y - c_r)^{d_r}$. Moreover, we have $\bar{f^*} = (Y - c_1)^{d_1}(Y - c_2)^{d_2} \cdots (Y - c_{r-1})^{d_{r-1}}$. The inductive hypothesis applied to f^* implies the existence of $f_1, \ldots, f_{r-1}$. ☐

2.3 Parallel Patterns

We are concerned with *thread-level parallelism*, where multiple threads of execution within a single process enable concurrent processing. Our parallel implementation employs several so-called *parallel patterns*—algorithmic structures and organizations for efficient parallel processing. We review a few patterns: *map*, *producer-consumer*, and *pipeline*. See [17] for a detailed discussion.

Map. The map pattern applies a function to each item in a collection, simultaneously executing the function on each independent data item. Often, the application of a map produces a new collection with the same shape as the input collection. Alternatively, the map pattern may modify each data item in place or, when combined with the *reduce* pattern, produce a single data item. The reduce pattern combines data items pair-wise using some *combiner* function.

When data items to be processed outnumber available threads, the map pattern can be applied block-wise, where the data collection is (evenly) partitioned and each thread assigned a partition rather than a single data item.

Where a **for** loop has independent iterations, the map pattern is easily applied to execute each iteration of the loop concurrently. Due to this ubiquity, the map pattern is often implicit with such parallel for loops simply being labelled **parallel_for**. In this way, the number of threads to use and the partitioning of the data collection can be a dynamic property of the algorithm.

Producer-Consumer and Asynchronous Generators. The producer-consumer pattern describes two functions connected by a queue. The producer creates data items, pushing them to the queue, meanwhile the consumer processes data items, pulling them from the queue. Where both the creation of data and its processing requires substantial work, producer and consumer may operate concurrently, with the queue providing inter-thread communication.

A *generator* or *iterator* is a special kind of co-routine function which **yield**s data elements one at a time, rather than many together as a collection; see, e.g. [23, Ch. 8]. Combining the producer-consumer pattern with generators allows for an *asynchronous generator*, where the generator function is the producer and the calling function is the consumer. The intermediary queue allows the generator to produce items meanwhile the calling function processes them.

Pipeline. The pipeline pattern is a sequence of stages, where the output of one stage is used as the input to another. Two consecutive stages form a producer-consumer pair, with internal stages being both a consumer and a producer. Concurrency arises where each stage of the pipeline may be executed in parallel. Moreover, the pipeline pattern allows for earlier data items to flow from one stage to the next without waiting for later items to become available.

In terms of the latency of processing a single data item, a pipeline does not improve upon its serial counterpart. Rather, a parallel pipeline improves throughput, the amount of data that can be processed in a given amount of time. Throughput is limited by the slowest stage of a pipeline, and thus special care must be given to ensure each stage of the pipeline runs in nearly equal time.

A pipeline may be implicitly and dynamically created where an asynchronous generator consumes data from another asynchronous generator. The number of asynchronous generator calls, and thus the number of stages in the pipeline, can be dynamic to fit the needs of the application at runtime.

3 Lazy Power Series

As we have seen in Section 2.1, certain arithmetic operations on power series naturally lead to a lazy evaluation scheme. In this scheme, homogeneous parts of a power series are computed one at a time for increasing degree, as requested. Our serial implementation of lazy power series is detailed in [8]. The underlying implementation of (sparse multivariate) polynomial arithmetic is that of [4] (indeed, dense multivariate arithmetic could prove beneficial, but that is left to future work). For the remainder of this paper, it is sufficient to understand that lazy power series rely on the following three principles:

(i) an update function to compute the homogeneous part of a given degree;

(ii) capturing of parameters required for that update function; and

(iii) storage of previously computed homogeneous parts.

Where a power series is constructed from arithmetic operations on other power series, the latter may be called the *ancestors* of the former. For example, the power series $f = g\,h$ has ancestors g and h and an update function $f_{(k)} = \sum_{i=0}^{k} g_{(i)} h_{(k-i)}$. In implementation, and in the algorithms which follow in this paper, we can thus augment a power series with: (i) its current precision; (ii) references to its ancestors, if any; and (iii) a reference to its update function.

Under this scheme, we make three remarks. Firstly, a power series can be lazily constructed using essentially no work. Indeed, the initialization of a lazy power series only requires specifying the appropriate update function and storing references to its ancestors. Secondly, specifying an update function and the ancestors of a power series is sufficient for defining and computing that power series. Thirdly, when updating a particular power series, its ancestors can automatically and recursively be updated as necessary using their own update functions.

Hence, it is sufficient to simply define the update function of a power series. For example, Algorithm 1 simultaneously updates p and α as produced from a Weierstrass preparation. Further, operations on power series should be understood to be only the initialization of a power series, with no terms of the power series yet computed; e.g., Algorithm 3 for Hensel factorization.

4 Algorithms and Complexity

In this section we present algorithms for Weierstrass preparation and Hensel factorization adapted from their constructive proofs; see Section 2. For each algorithm we analyze its complexity. The algorithms—and eventual parallel variations and implementations, see Sections 5–6—are presented for the general multivariate case, with only the complexity estimates limited to the bivariate case. These results culminate as Theorem 4.3 and Corollary 4.8, which respectively give the overall complexity of our algorithms for WPT and Hensel factorization. Meanwhile, Observation 4.2, Corollary 4.4, and Theorem 4.6 more closely analyze the distribution of work to guide and load-balance our parallel algorithms.

4.1 Weierstrass Preparation

From the proof of Weierstrass preparation (Theorem 2.3), we derive WEIERSTRASSUPDATE (Algorithm 1). That proof proceeds modulo increasing powers of the maximal ideal $\mathcal{M}$, which is equivalent to computing homogeneous parts of increasing degree, just as required for our lazy power series. For an application of Weierstrass preparation producing p and α, this WEIERSTRASSUPDATE acts as the update function for p and α, updating both simultaneously.

Algorithm 1 WEIERSTRASSUPDATE(k, f, p, α)

Input: $f = \sum_{i=0}^{d+m} a_i Y^i$, $p = Y^d + \sum_{i=0}^{d-1} b_i Y^i$, $\alpha = \sum_{i=0}^{m} c_i Y^i$, $a_i, b_i, c_i \in \mathbb{K}[[X_1, \ldots, X_n]]$ satisfying Theorem 2.3, with $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ known modulo $\mathcal{M}^k$, $\mathcal{M}$ the maximal ideal of $\mathbb{K}[[X_1, \ldots, X_n]]$.

Output: $b_0, \ldots, b_{d-1}, c_0, \ldots, c_m$ known modulo $\mathcal{M}^{k+1}$, updated in-place.

```
for $i := 0$ to $d-1$ do                                              ▷ phase 1
    $F_{i\,(k)} := a_{i\,(k)}$
    if $i \leq m$ then
        for $j := 0$ to $i-1$ do
            $F_{i\,(k)} := F_{i\,(k)} - (b_j\, c_{i-j})_{(k)}$
    else
        for $j := 0$ to $m-1$ do
            $F_{i\,(k)} := F_{i\,(k)} - (b_{i+j-m}\, c_{m-j})_{(k)}$
    $s := 0$
    for $j := 1$ to $k-1$ do
        $s := s + b_{i\,(k-j)} \times c_{0\,(j)}$
    $b_{i\,(k)} := \left(F_{i\,(k)} - s\right) / c_{0\,(0)}$
$c_{m\,(k)} := a_{d+m\,(k)}$                                          ▷ phase 2
for $i := 1$ to $m$ do
    if $i \leq d$ then
        $c_{m-i\,(k)} := a_{d+m-i\,(k)} - \sum_{j=1}^{i} (b_{d-j} c_{m-i+j})_{(k)}$
    else
        $c_{m-i\,(k)} := a_{d+m-i\,(k)} - \sum_{j=1}^{d} (b_{d-j} c_{m-i+j})_{(k)}$
```

By rearranging the first d equations of (1) and applying Lemma 2.2 we obtain "phase 1" of WEIERSTRASSUPDATE, where each coefficient of p is updated. By rearranging the next $m+1$ equations of (1) we obtain "phase 2" of WEIERSTRASSUPDATE, where each coefficient of α is updated. From Algorithm 1, it is then routine to show the following two observations, which lead to Theorem 4.3.

Observation 4.1 (Weierstrass phase 1 complexity) *For* WEIERSTRASSUPDATE *over* $\mathbb{K}[[X_1]][Y]$*, computing* $b_{i(k)}$*, for* $0 \le i < d$*, requires* $2ki + 2k - 1$ *operations in* $\mathbb{K}$ *if* $i \le m$*, or* $2km + 2k - 1$ *operations in* $\mathbb{K}$ *if* $i > m$*.*

Observation 4.2 (Weierstrass phase 2 complexity) *For* WEIERSTRASSUPDATE *over* $\mathbb{K}[[X_1]][Y]$*, computing* $c_{m-i(k)}$*, for* $0 \le i < m$*, requires* $2ki$ *operations in* $\mathbb{K}$ *if* $i \le d$*, or* $2kd$ *operations in* $\mathbb{K}$ *if* $i > d$*.*

Theorem 4.3 (Weierstrass preparation complexity) *Weierstrass preparation producing* $f = p\,\alpha$*, with* $f, p, \alpha \in \mathbb{K}[[X_1]][Y]$*,* $\deg(p) = d$*,* $\deg(\alpha) = m$*, requires* $dmk^2 + dk^2 + dmk$ *operations in* $\mathbb{K}$ *to compute* p *and* α *to precision* k*.*

PROOF. Let i be the index of a coefficient of p or α. Consider the cost of computing the homogeneous part of degree k of each coefficient of p and α. First consider $i < t = \min(d, m)$. From Observations 4.1 and 4.2, computing the kth homogeneous part of each b_i and c_i respectively requires $2ki + 2k - 1$ and $2ki$ operations in $\mathbb{K}$. For $0 \le i < t$, this yields a total of $2kt^2 + 2kt - t$. Next, we have three cases: (a) $t = d = m$, (b) $m = t < i < d$, or (c) $d = t < i < m$. In case (a) there is no additional work. In case (b), phase 1 contributes an additional $(d-m)(2km+2k-1)$ operations. In case (c), phase 2 contributes an additional $(m-d)(2kd)$ operations. In all cases, the total number of operations to update p and α from precision $k-1$ to precision k is $2dmk + 2dk - d$. Finally, to compute p and α up to precision k requires $dmk^2 + dk^2 + dmk$ operations in $\mathbb{K}$. □

A useful consideration is when the input to Weierstrass preparation is monic; this arises for each application of WPT in Hensel factorization. Then, α is necessarily monic, and the overall complexity of Weierstrass preparation is reduced. In particular, we save computing $(b_{i-m}c_m)_{(k)}$ for the update of b_i, $i \ge m$ (Algorithm 1, Line 8), and save computing $(b_{d-i}c_m)_{(k)}$ for the update of each c_{m-i}, $i \le d$ (Algorithm 1, Line 16). The following corollary states this result.

Corollary 4.4 (Weierstrass preparation complexity for monic input) *Weierstrass preparation producing* $f = p\,\alpha$ *with* $f, p, \alpha \in \mathbb{K}[[X_1]][Y]$*,* f *monic in* Y*,* $\deg(p) = d$ *and* $\deg(\alpha) = m$*, requires* $dmk^2 + dmk$ *operations in* $\mathbb{K}$ *to compute* p *and* α *up to precision* k*.*

4.2 Hensel Factorization

Before we begin Hensel factorization, we will first see how to perform a translation, or Taylor shift, by lazy evaluation. For $f = \sum_{i=0}^{d} a_i Y^i \in \mathbb{K}[[X_1, \ldots, X_n]][Y]$ and $c \in \mathbb{K}$, computing $f(Y + c)$ begins by pre-computing the coefficients of the binomial expansions $(Y+c)^j$ for $0 \le j \le d$. These coefficients are stored in a matrix $\mathbf{S}$. Then, each coefficient of $f(Y + c) = \sum_{i=0}^{d} b_i Y^i$ is a linear combination of the coefficients of f scaled by the appropriate elements of $\mathbf{S}$. Since those elements of $\mathbf{S}$ are only elements of $\mathbb{K}$, this linear combination does not change the degree and, for some integer k, $b_{i(k)}$ relies only on $a_{\ell(k)}$ for $i \le \ell \le d$. This method is described in Algorithm 2; and its complexity is easily stated as Observation 4.5.

Algorithm 2 TaylorShiftUpdate(k, f, $\mathbf{S}$, i)

```
Input: For $f = \sum_{j=0}^{d} a_j Y^j$, $g = f(Y + c) = \sum_{j=0}^{d} b_j Y^j$, obtain the homogeneous part of degree $k$ for $b_i$. $\mathbf{S} \in \mathbb{K}^{(d+1)\times(d+1)}$ is a lower triangular matrix of coefficients of $(Y + c)^j$ for $j = 0, \ldots, d$,
Output: $b_{i_{(k)}}$, the homogeneous part of degree $k$ of $b_i$.
$b_{i_{(k)}} := 0$
for $\ell := i$ to $d$ do
    $j := \ell + 1 - i$
    $b_{i_{(k)}} := b_{i_{(k)}} + S_{\ell+1,j} \times a_{\ell(k)}$
return $b_{i_{(k)}}$
```

Algorithm 3 HenselFactorization(f)

```
Input: $f = Y^d + \sum_{i=0}^{d-1} a_i Y^i, a_i \in \mathbb{K}[[X_1, \ldots, X_n]]$.
Output: $f_1, \ldots, f_r$ satisfying Theorem 2.4.
$\bar{f} = f(0, \ldots, 0, Y)$
$(c_1, \ldots, c_r), (d_1, \ldots, d_r)$ := roots and their multiplicities of $\bar{f}$
$c_1, \ldots, c_r$ := sort($[c_1, \ldots, c_r]$) by increasing multiplicity        ▷ see Theorem 4.6
$\widehat{f}_1 := f$
for $i := 1$ to $r - 1$ do
    $g_i := \widehat{f}_i(Y + c_i)$
    $p_i, \alpha_i$ := WeierstrassPreparation($g$)
    $f_i := p_i(Y - c_i)$
    $\widehat{f}_{i+1} := \alpha_i(Y - c_i)$
$f_r := \widehat{f}_r$
return $f_1, \ldots, f_r$
```

$$\begin{tikzcd}
 & & p_1 \arrow[r, "-c_1"] & f_1 & & p_2 \arrow[r, "-c_2"] & f_2 & & p_3 \arrow[r, "-c_3"] & f_3 \\
f \arrow[r, "+c_1"] & g_1 \arrow[ur] \arrow[r] & \alpha_1 \arrow[r, "-c_1"] & \widehat{f}_2 \arrow[r, "+c_2"] & g_2 \arrow[ur] \arrow[r] & \alpha_2 \arrow[r, "-c_2"] & \widehat{f}_3 \arrow[r, "+c_3"] & g_3 \arrow[ur] \arrow[r] & \alpha_3 \arrow[r, "-c_3"] & f_4
\end{tikzcd}$$

Figure 1: The ancestor chain for the Hensel factorization $f = f_1 f_2 f_3 f_4$. Updating f_1 requires updating g_1, p_1, α_1; then updating f_2 requires updating $\widehat{f}_2, g_2, p_2, \alpha_2$; then updating f_3 requires updating $\widehat{f}_3, g_3, p_3, \alpha_3$; then updating f_4 requires only its own Taylor shift. These groupings form the eventual stages of the Hensel pipeline (Algorithm 8).

Observation 4.5 (Taylor shift complexity) *For a UPoPS $f = \sum_{i=0}^{d} a_i Y^i \in \mathbb{K}[[X_1]][Y]$, computing the homogeneous part of degree k for all coefficients of the shifted UPoPS $f(Y + c)$ requires $d^2 + 2d + 1$ operations in $\mathbb{K}$.*

Having specified the update functions for WPT and Taylor shift, lazy Hensel factorization is immediate, requiring only the appropriate chain of ancestors. Algorithm 3 shows this initialization through repeated applications of Taylor shift and Weierstrass preparation. Note that factors are sorted by increasing degree to enable better load-balance in the eventual parallel algorithm. Fig. 1 shows the chain of ancestors created by $f = f_1 f_2 f_3 f_4$ and the grouping of ancestors required to update each factor; the complexity of which is shown in Theorem 4.6. Corollary 4.7 follows immediately and Corollary 4.8 gives the total complexity of Hensel factorization. Here, we ignore the initial cost of factorizing $\bar{f}$.

Theorem 4.6 (Hensel factorization complexity per factor) *Let $\widehat{d}_i$ be the degree of $\widehat{f}_i$ during* HenselFactorization *applied to $f \in \mathbb{K}[[X_1]][Y]$, $\deg(f) = d$. To* *update* f_1, $\deg(f_1) = d_1$ *to precision* k *requires* $d_1\widehat{d}_2k^2 + d^2k + d_1dk + 2d_1k + 2dk + 2k$ *operations in* $\mathbb{K}$. *To update* f_i, $\deg(f_i) = d_i$, *for* $1 < i < r$, *to precision* k *requires* $d_i\widehat{d}_{i+1}k^2 + 2\widehat{d}_i^{\,2}k + d_i\widehat{d}_ik + 2d_ik + 4\widehat{d}_ik + 3k$ *operations in* $\mathbb{K}$. *To update* f_r, $\deg(f_r) = d_r$, *to precision* k *requires* $d_r^2k + 2d_rk + k$ *operations in* $\mathbb{K}$.

PROOF. Updating the first factor produced by HENSELFACTORIZATION requires one Taylor shift of degree d, one Weierstrass preparation producing p_1 and α_1 of degree d_1 and $\widehat{d}_2 = d - d_1$, and one Taylor shift of degree d_1 to obtain f_1 from p. From Observation 4.5 and Corollary 4.4 we have that the Taylor shifts require $k(d^2 + 2d + 1) + k(d_1^2 + 2d_1 + 1)$ operations in $\mathbb{K}$ and the Weierstrass preparation requires $d_1(d - d_1)k^2 + d_1(d - d_1)k$ operations in $\mathbb{K}$. The total cost counted as operations in $\mathbb{K}$ is thus $d_1\widehat{d}_2k^2 + d^2k + d_1dk + 2d_1k + 2dk + 2k$.

Updating each following factor, besides the last, requires one Taylor shift of degree $\widehat{d}_i$ to update $\widehat{f}_i$ from α_{i-1}, one Taylor shift of degree $\widehat{d}_i$ to update g_i from $\widehat{f}_i$, one Weierstrass preparation to obtain p_i and α_i of degree d_i and $\widehat{d}_{i+1} = \widehat{d}_i - d_i$, and one Taylor shift of degree d_i to obtain f_i from p_i. The Taylor shifts require $2k(\widehat{d}_i^{\,2} + 2\widehat{d}_i + 1) + k(d_i^2 + 2d_i + 1)$ operations in $\mathbb{K}$. The Weierstrass preparation requires $d_i(\widehat{d}_i - d_i)k^2 + d_i(\widehat{d}_i - d_i)k$ operations in $\mathbb{K}$. The total cost counted as operations in $\mathbb{K}$ is thus $d_i\widehat{d}_{i+1}k^2 + 2\widehat{d}_i^{\,2}k + d_i\widehat{d}_ik + 2d_ik + 4\widehat{d}_ik + 3k$.

Finally, updating the last factor to precision k requires a single Taylor shift of degree d_r costing $d_r^2k + 2d_rk + k$ operations in $\mathbb{K}$. $\square$

Corollary 4.7 (Hensel factorization complexity per iteration) *Let* $\widehat{d}_i$ *be the degree of* $\widehat{f}_i$ *during the* HENSELFACTORIZATION *algorithm applied to* $f \in \mathbb{K}[[X_1]][Y]$, $\deg(f) = d$. *Computing the kth homogeneous part of* f_1, $\deg(f_1) = d_1$, *requires* $2d_1\widehat{d}_2k + d_1^2 + d^2 + 2d_1 + 2d + 2$ *operations in* $\mathbb{K}$. *Computing the kth homogeneous part of* f_i, $\deg(f_i) = d_i$, $1 < i < r$, *requires* $2d_i\widehat{d}_{i+1}k + d_i^2 + 2\widehat{d}_i^{\,2} + 4\widehat{d}_i + 2d_i + 3$ *operations in* $\mathbb{K}$. *Computing the kth homogeneous part of* f_r, $\deg(f_r) = d_r$, *requires* $d_r^2 + 2d_r + 1$ *operations in* $\mathbb{K}$.

Corollary 4.8 (Hensel factorization complexity) HENSELFACTORIZATION *producing* $f = f_1 \cdots f_r$, *with* $f \in \mathbb{K}[[X_1]][Y]$, $\deg(f) = d$, *requires* $\mathcal{O}(d^3k + d^2k^2)$ *operations in* $\mathbb{K}$ *to update all factors to precision* k.

PROOF. Let $f_1, \ldots, f_r$ have respective degrees $d_1, \ldots, d_r$. Let $\widehat{d}_i = \sum_{j=i}^{r} d_j$ (thus $\widehat{d}_1 = d$ and $\widehat{d}_r = d_r$). From Theorem 4.6, each f_i, $1 \le i < r$ requires $\mathcal{O}(d_i\widehat{d}_{i+1}k^2 + \widehat{d}_i^{\,2}k)$ operations in $\mathbb{K}$ to be updated to precision k (or $\mathcal{O}(d_r^2k)$ for f_r). We have $\sum_{i=1}^{r-1} d_i\widehat{d}_{i+1} \le \sum_{i=1}^{r-1} d_id < d^2$ and $\sum_{i=1}^{r} \widehat{d}_i^{\,2} \le \sum_{i=1}^{r} d^2 = rd^2 \le d^3$. Hence, all factors can be updated to precision k within $\mathcal{O}(d^3k + d^2k^2)$ operations in $\mathbb{K}$. $\square$

Corollary 4.8 shows that the two dominant terms in the cost of computing a Hensel factorization of a UPoPS of degree d, up to precision k, are d^3k and d^2k^2. From the proof of Theorem 4.6, the former term arises from the cost of the Taylor shifts in Y, meanwhile, the latter term arises from the (polynomial) multiplication of homogeneous parts in Weierstrass preparation. This observation then leads to the following conjecture. Recall that $M(n)$ denotes a polynomial multiplication time [28, Ch. 8]. From [26], relaxed algorithms, which improve the performance of lazy evaluation schemes, can be used to compute a power series product in $\mathbb{K}[[X_1]]$ up to precision k in at most $\mathcal{O}(M(k)\log k)$ operations in $\mathbb{K}$ (or less, in view of the improved relaxed multiplication of [27]).

Conjecture 4.9 *Let* $f \in \mathbb{K}[[X_1]][Y]$ *factorize as* $f_1 \cdots f_r$ *using* HENSELFACTORIZATION. *Let* $\deg(f) = d$. *Updating the factors* $f_1, \ldots, f_r$ *to precision* k *using relaxed algorithms requires at most* $\mathcal{O}(d^3 k + d^2 M(k)\log k)$ *operations in* $\mathbb{K}$.

Comparatively, the Hensel–Sasaki Construction requires at most $\mathcal{O}(d^3 M(d) + dM(d)k^2)$ operations in $\mathbb{K}$ to compute the first k terms of all factors of $f \in \mathbb{K}[X_1, Y]$, where f has total degree d [1]. The method of Kung and Traub [15], requires $\mathcal{O}(d^2 M(k))$. Already, Corollary 4.8—where $d = \deg(f, Y)$—shows that our Hensel factorization is an improvement on Hensel–Sasaki (d^2k^2 versus $dM(d)k^2$). If Conjecture 4.9 is true, then Hensel factorization can be within a factor of $\log k$ of Kung and Traub's method. Nonetheless, this conjecture is highly encouraging where $k \gg d$, particularly where we have already seen that our current, suboptimal, method performs better in practice than Hensel–Sasaki and the method of Kung and Traub [8]. Proving this conjecture is left to future work.

5 Parallel Algorithms

Section 4 presented lazy algorithms for Weierstrass preparation, Taylor shift, and Hensel factorization. It also presented complexity estimates for those algorithms. Those estimates will soon be used to help dynamically distribute hardware resources (threads) in a parallel variation of Hensel factorization; in particular, a Hensel factorization pipeline where each pipeline stage updates one or more factors, see Algorithms 7–9. But first, we will examine parallel processing techniques for Weierstrass preparation.

5.1 Parallel Algorithms for Weierstrass Preparation

Algorithm 1 shows that p and α from a Weierstrass preparation can be updated in two phases: p in phase 1, and α in phase 2. Ultimately, these updates rely on the computation of the homogeneous part of some power series product. Algorithm 4 presents a simple map-reduce pattern (see Section 2.3) for computing such a homogeneous part. Moreover, this algorithm is designed such that, recursively, all ancestors of a power series product are also updated using parallelism. Note that UPDATETODEGPARALLEL called on a UPoPS simply recurses on each of its coefficients.

Using the notation of Algorithm 1, recall that, e.g., $F_i := a_i - \sum_{j=0}^{i-1}(b_j c_{i-j})$, for $i \leq m$. Using lazy power series arithmetic, this entire formula can be encoded by a chain of ancestors, and one simply needs to update F_i to trigger a cascade of updates through its ancestors. In particular, using Algorithm 4, the homogeneous part of each product $b_j c_{i-j}$ is recursively computed using map-reduce. Similarly, Lemma 2.2 can be implemented using map-reduce (see Algorithm 5) to replace Lines 9–12 of Algorithm 1. Phase 1 of Weierstrass, say WEIERSTRASSPHASE1PARALLEL,

Algorithm 4 UpdateToDegParallel(k, f, t)

```
Input: A positive integer $k$, $f \in \mathbb{K}[[X_1, \ldots, X_n]]$ known to at least precision $k-1$. If $f$ has ancestors,
    it is the result of a binary operation. A positive integer $t$ for the number of threads to use.
Output: $f$ is updated to precision $k$, in place.
if $f_{(k)}$ already computed then
    return
$g$, $h$ := FirstAncestor($f$), SecondAncestor($f$)
UpdateToDegParallel($k$, $g$, $t$);
UpdateToDegParallel($k$, $h$, $t$);
if $f$ is a product then
    $\mathcal{V} := [0, \ldots, 0]$                                ▷ 0-indexed list of size $t$
    parallel_for $j := 0$ to $t-1$
        for $i := {}^{jk}/_{t}$ to ${}^{(j+1)k}/_{t} - 1$ while $i \leq k$ do
            $\mathcal{V}[j] := \mathcal{V}[j] + g_{(i)} h_{(k-i)}$
    $f_{(k)} := \sum_{j=0}^{t-1} \mathcal{V}[j]$                    ▷ reduce
else if $f$ is a $p$ from a Weierstrass preparation then
    WeierstrassPhase1Parallel($k$,$g$,$f$,$h$,WeierstrassData($f$),$t$)
else if $f$ is an $\alpha$ from a Weierstrass preparation then
    WeierstrassPhase2Parallel($k$, $g$, $h$, $f$, $t$)
else
    UpdateToDeg($k$, $f$)
```

Algorithm 5 LemmaForWeierstrass(k, f, g, h, t)

```
Input: $f, g, h \in \mathbb{K}[[X_1, \ldots, X_n]]$ such that $f = gh$, $f_{(0)} = 0$, $h_{(0)} \neq 0$, $f$ known to precision $k$, and $g, h$
    known to precision $k-1$. $t \geq 1$ the number of threads to use.
Output: $g_{(k)}$.
$\mathcal{V} := [0, \ldots, 0]$                                    ▷ 0-indexed list of size $t$
parallel_for $j := 0$ to $t-1$
    for $i := {}^{jk}/_{t} + 1$ to ${}^{(j+1)k}/_{t}$ while $i < k$ do
        $\mathcal{V}[j] := \mathcal{V}[j] + g_{(k-i)} h_{(i)}$
end for
return $\left(f_{(k)} - \sum_{j=0}^{t-1} \mathcal{V}[j]\right) / h_{(0)}$
```

thus reduces to a loop over i from 0 to $d-1$, calling Algorithm 4 to update F_i to precision k, and calling Algorithm 5 to compute $b_{i(k)}$.

Algorithm 4 uses several simple subroutines: FirstAncestor and SecondAncestor gets the first and second ancestor of a power series, WeierstrassData gets a reference to the list of F_i's, and UpdateToDeg calls the serial update function of a lazy power series to ensure its precision is at least k; see Section 3.

Now consider phase 2 of WeierstrassUpdate. Notice that computing the homogeneous part of degree k for c_{m-i}, $0 \leq i \leq m$ only requires each c_{m-i} to be known up to precision $k-1$, since each $b_j \in \mathcal{M}$ for $0 \leq j < d$. This implies that the phase 2 **for** loop of WeierstrassUpdate has independent iterations. We thus apply the map pattern directly to this loop itself, rather than relying on the map-reduce pattern of UpdateToDegParallel. However, consider the following two facts: the cost of computing each c_{m-i} is different (Observation 4.2 and Corollary 4.4), and, for a certain number of available threads t, it may be impossible to partition the iterations of the loop into t partitions of equal work. Yet, partitioning the loop itself is preferred for greater parallelism.

Hence, for phase 2, a dynamic decision is made to either apply the map pattern to the loop over c_{m-i}, or to apply the map pattern within UpdateToDegParallel for each c_{m-i}, or both. This decision process is detailed in Algorithm 6, where t partitions of equal work try to be found to apply the map pattern to only the loop itself. If unsuccessful, ${}^{t}/_{2}$ partitions of equal work try to be found, with 2 threads to be used within UpdateToDegParallel of each partition. If that, too,

Algorithm 6 WeierstrassPhase2Parallel(k, f, p, α, t)

Input: $f = \sum_{i=0}^{d+m} a_i Y^i$, $p = Y^d + \sum_{i=0}^{d-1} b_i Y^i$, $\alpha = \sum_{i=0}^{m} c_i Y^i$, $a_i, b_i, c_i \in \mathbb{K}[[X_1, \dots, X_n]]$ satisfying Theorem 2.3. $b_0, \dots, b_{d-1}$ known modulo $\mathcal{M}^{k+1}$, $c_0, \dots, c_m$ known modulo $\mathcal{M}^k$, for $\mathcal{M}$ the maximal ideal of $\mathbb{K}[[X_1, \dots, X_n]]$. $t \geq 1$ for the number of threads to use.

Output: $c_0, \dots, c_m$ known modulo $\mathcal{M}^{k+1}$, updated in-place.

```
 1: $work := 0$
 2: for $i := 1$ to $m$ do                          ▷ estimate work using Observation 4.2, Corollary 4.4
 3:     if $i \leq d$ then $work := work + i - (a_{d+m} = 0)$        ▷ eval. Boolean as an integer
 4:     else $work := work + d$
 5: $t' := 1$; $targ := work \,/\, t$
 6: $work := 0$; $j := 1$
 7: $\mathcal{I} := [-1, 0, \dots, 0]$                                 ▷ 0-indexed list of size $t + 1$
 8: for $i := 1$ to $m$ do
 9:     if $i \leq d$ then $work := work + i - (a_{d+m} = 0)$
10:     else $work := work + d$
11:     if $work \geq targ$ then
12:         $\mathcal{I}[j] := i$; $work := 0$; $j := j + 1$
13: if $j \leq t$ and $t' < 2$ then              ▷ work did not distribute evenly; try again with $t = t/2$
14:     $t := t \,/\, 2$; $t' := 2$
15:     goto Line 6
16: else if $j \leq t$ then                     ▷ still not even, use all threads in UpdateToDegParallel
17:     $\mathcal{I}[1] := m$; $t' := 2t$; $t := 1$
18: parallel_for $\ell := 1$ to $t$
19:     for $i := \mathcal{I}[\ell - 1] + 1$ to $\mathcal{I}[\ell]$ do
20:         UpdateToDegParallel($k$, $c_{m-i}$, $t'$)
```

Algorithm 7 HenselPipelineStage(k, f_i, t, gen)

Input: A positive integer k, $f_i = Y^{d_i} + \sum_{i=0}^{d_i - 1} a_i Y^i, a_i \in \mathbb{K}[[X_1, \dots, X_n]]$. A positive integer t the number of threads to use within this stage. gen a generator for the previous pipeline stage.

Output: a sequence of integers j signalling f_i is known to precision j. This sequence ends with k.

```
$p :=$ Precision($f_i$)                             ▷ get the current precision of $f_i$
do
    $k' :=$ gen()                         ▷ A blocking function call until gen yields
    for $j := p$ to $k'$ do
        UpdateToDegParallel($j$, $f_i$, $t$)
        yield $j$
    $p := k'$
while $k' < k$
```

is unsuccessful, then each c_{m-i} is updated one at a time using the total number of threads t within UpdateToDegParallel.

5.2 Parallel Algorithms for Hensel Factorization

Let $f = f_1 \cdots f_r$ be a Hensel factorization where the factors have respective degrees $d_1, \dots, d_r$. From Algorithm 3 and Figure 1, we have already seen that the repeated applications of Taylor shift and Weierstrass preparation naturally form a chain of ancestors, and thus a pipeline. Using the notation of Algorithm 3, updating f_1 requires updating g_1, p_1, α_1. Then, updating f_2 requires updating $\widehat{f}_2, g_2, p_2, \alpha_2$, and so on. These groups easily form stages of a pipeline, where updating f_1 to degree $k - 1$ is a prerequisite for updating f_2 to degree $k - 1$. Moreover, meanwhile f_2 is being updated to degree $k-1$, f_1 can simultaneously be updated to degree k. Such a pattern holds for all successive factors.

Algorithms 7 and 8 show how the factors of a Hensel factorization can all be simultaneously updated to degree k using asynchronous generators, denoted by the constructor AsyncGenerator, forming the so-called *Hensel pipeline*. Algorithm 7 shows a single pipeline stage as an asynchronous generator, which itself consumes

Algorithm 8 HenselFactorizationPipeline(k, $\mathcal{F}$, $\mathcal{T}$)

```
Input: A positive integer $k$, $\mathcal{F} = \{f_1, \ldots, f_r\}$, the output of HenselFactorization. $\mathcal{T} \in \mathbb{Z}^r$ a 0-indexed
    list of the number of threads to use in each stage, $\mathcal{T}[r-1] > 0$.
Output: $f_1, \ldots, f_r$ updated in-place to precision $k$.
gen := ( ) → {yield $k$}                         ▷ An anonymous function asynchronous generator
for $i := 0$ to $r - 1$ do
    if $\mathcal{T}[i] > 0$ then
        ▷ Capture HenselPipelineStage($k$, $f_{i+1}$, $\mathcal{T}[i]$, gen) as a
          function object, passing the previous gen as input
        gen := AsyncGenerator(HenselPipelineStage, $k$, $f_{i+1}$, $\mathcal{T}[i]$, gen)
do
    $k'$ := gen()                                ▷ ensure last stage completes before returning
while $k' < k$
```

data from another asynchronous generator—just as expected from the pipeline pattern. Algorithm 8 shows the creation, and joining in sequence, of those generators. The key feature of these algorithms is that a generator (say, stage i) produces a sequence of integers (j) which signals to the consumer (stage $i+1$) that the previous factor has been computed up to precision j and thus the required data is available to update its own factor to precision j.

Notice that Algorithm 8 still follows our lazy evaluation scheme. Indeed, the factors are updated all at once up to precision k, starting from their current precision. However, for optimal performance, the updates should be applied for large increases in precision, rather than repeatedly increasing precision by one.

Further considering performance, Theorem 4.6 showed that the cost for updating each factor of a Hensel factorization is different. In particular, for $\widehat{d_i} := \sum_{j=i}^{r} d_j$, updating factor f_i scales as $d_i \widehat{d_{i+1}} k^2$. The work for each stage of the proposed pipeline is unequal and the pipeline is unlikely to achieve good parallel speedup. However, Corollary 4.7 shows that the work ratios between stages do not change for increasing k, and thus a static scheduling scheme is sufficient.

Notice that Algorithm 7 takes a parameter t for the number of threads to use internally. As we have seen in Section 5.1, the Weierstrass update can be performed in parallel. Consequently, each stage of the Hensel pipeline uses t threads to exploit such parallelism. We have thus composed the two parallel schemes, applying map-reduce within each stage of the parallel pipeline. This composition serves to load-balance the pipeline. For example, the first stage may be given t_1 threads and the second stage given t_2 threads, with $t_1 > t_2$, so that the two stages may execute in nearly equal time.

To further encourage load-balancing, each stage of the pipeline need not update a single factor, but rather a group of successive factors. Algorithm 9 applies Theorem 4.6 to attempt to load-balance each stage s of the pipeline by assigning a certain number of threads t_s and a certain group of factors $f_{s_1}, \ldots, f_{s_2}$ to it. The goal is for $\sum_{i=s_1}^{s_2} d_i \widehat{d_{i+1}} \,/\, t_s$ to be roughly equal for each stage.

6 Experimentation and Discussion

The previous section introduced parallel schemes for Weierstrass preparation and Hensel factorization based on the composition of the map-reduce and pipeline parallel patterns. Our lazy power series and parallel schemes have been implemented in C/C++ as part of the Basic Polynomial Algebra Subprograms (BPAS) library [2].

Algorithm 9 DistributeResourcesHensel($\mathcal{F}$, t_{tot})

```
Input: $\mathcal{F} = \{f_1, \ldots, f_r\}$ the output of HenselFactorization. $t_{tot} > 1$ the total number of threads.
Output: $\mathcal{T}$, a list of size $r$, where $\mathcal{T}[i]$ is the number of threads to use for updating $f_{i+1}$. The number
    of positive entries in $\mathcal{T}$ determines the number of pipeline stages. $\mathcal{T}[i] = 0$ encodes that $f_{i+1}$ should
    be computed within the same stage as $f_{i+2}$.
$\mathcal{T} := [0, \ldots, 0, 1]$; $t := t_{tot} - 1$                ▷ $\mathcal{T}[r-1] = 1$ ensures last factor gets updated
$d := \sum_{i=1}^{r} \deg(f_i)$
$\mathcal{W} := [0, \ldots, 0]$                                        ▷ A 0-indexed list of size $r$
for $i := 1$ to $r - 1$ do
    $\mathcal{W}[i-1] := \deg(f_i)(d - \deg(f_i))$                      ▷ Estimate work by Theorem 4.6, $d_i \widehat{d}_{i+1}$
    $d := d - \deg(f_i)$
$totalWork := \sum_{i=0}^{r-1} \mathcal{W}[i]$
$ratio := 0$; $targ := 1 \,/\, t$
for $i := 0$ to $r$ do
    $ratio := ratio + (\mathcal{W}[i] \,/\, totalWork)$
    if $ratio \geq targ$ then
        $\mathcal{T}[i] :=$ round$(ratio \cdot t)$; $ratio := 0$
$t := t_{tot} - \sum_{i=0}^{r-1} \mathcal{T}[i]$                       ▷ Give any excess threads to the earlier stages
for $i := 0$ to $r - 1$ while $t > 0$ do
    $\mathcal{T}[i] := \mathcal{T}[i] + 1$;  $t := t - 1$
return $\mathcal{T}$
```

These parallel algorithms are implemented using generic support for task parallelism, thread pools, and asynchronous generators, also provided in the BPAS library. The details of this parallel support are discussed in [5] and [6].

Our experimentation was collected on a machine running Ubuntu 18.04.4 with two Intel Xeon X5650 processors, each with 6 cores (12 cores total) at 2.67 GHz, and a 12x4GB DDR3 memory configuration at 1.33 GHz. All data shown is an average of 3 trials. BPAS was compiled using GMP 6.1.2 [12]. We work over $\mathbb{Q}$ as these examples do not require algebraic numbers to factor into linear factors. We thus borrow univariate integer polynomial factorization from NTL 11.4.3 [24]. Where algebraic numbers are required, the `MultivariatePowerSeries` package of Maple [3], an extension of our work in [8], is available.

We begin by evaluating Weierstrass preparation for two families of examples:

(*i*) $u_r = \sum_{i=2}^{r} (X_1^2 + X_2 + i)Y^i + (X_1^2 + X_2)Y + X_1^2 + X_1X_2 + X_2^2$

(*ii*) $v_r = \sum_{i=\lceil r/2 \rceil}^{r} (X_1^2 + X_2 + i)Y^i + \sum_{i=1}^{\lceil r/2 \rceil - 1} (X_1^2 + X_2)Y^i + X_1^2 + X_1X_2 + X_2^2$

Applying Weierstrass preparation to u_r results in p with degree 2. Applying Weierstrass preparation to v_r results in p with degree $\lceil r/2 \rceil$. Fig. 2 summarizes the resulting execution times and parallel speedups. Generally, speedup increases with increasing degree in Y and increasing precision computed.

Recall that parallelism arises in two ways: computing summations of products of homogeneous parts (the **parallel_for** loops in Algorithms 4 and 5), and the **parallel_for** loop over updating c_{m-i} in Algorithm 6. The former has an inherent limitation: computing a multivariate product with one operand of low degree and one of high degree is much easier than computing one where both operands are of moderate degree. Evenly partitioning the iterations of the loop does not result in even work per thread. This is evident in comparing the parallel speedup between u_r and v_r; the former, with higher degree in α, relies less on parallelism coming from those products. Better partitioning is needed and is left to future work.

We evaluate our parallel Hensel factorization using three families of problems:

(*i*) $x_r = \prod_{i=1}^{r} (Y - i) + X_1(Y^3 + Y)$

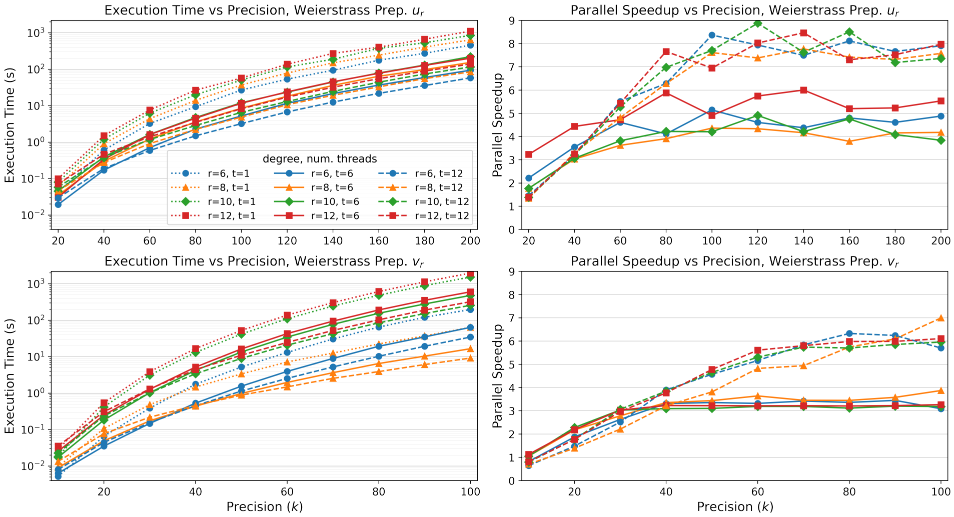

Figure 2: Comparing Weierstrass preparation of u_r and v_r for $r \in \{6, 8, 10, 12\}$ and number of threads $t \in \{1, 6, 12\}$. First column: execution time of u_r and v_r; second column: parallel speedup of u_r and v_r. Profiling of v_6 shows that its exceptional relative performance is attributed to remarkably good branch prediction.

(ii) $y_r = \prod_{i=1}^{r}(Y - i)^i + X_1(Y^3 + Y)$

(iii) $z_r = \prod_{i=1}^{r}(Y + X_1 + X_2 - i) + X_1X_2(Y^3 + Y)$

These families represent three distinct computational configurations: (i) factors of equal degree, (ii) factors of distinct degrees, and (iii) multivariate factors. The comparison between x_r and y_r is of interest in view of Theorem 4.6.

Despite the inherent challenges of irregular parallelism arising from stages with unequal work, the composition of parallel patterns allows for load-balancing between stages and the overall pipeline to achieve relatively good parallel speed-up. Fig. 3 summarizes these results while Table 1 presents the execution time per factor (or stage, in parallel). Generally speaking, potential parallelism increases with increasing degree and increasing precision.

The distribution of a discrete number of threads to a discrete number of pipeline stages is a challenge; a perfect distribution requires a fractional number of threads per stage. Nonetheless, in addition to the distribution technique presented in Algorithm 9, we can examine hand-chosen assignments of threads to stages. One can first determine the time required to update each factor in serial, say for some small k, and then use that time as the work estimates in Algorithm 9, rather than using the complexity estimates. This latter technique is depicted in Fig. 3 as *opt* and in Table 1 as Time-est. threads. This is still not perfect, again because of the discrete nature of threads, and the imperfect parallelization of computing summations of products of homogeneous parts.

In future, we must consider several important factors to improve performance. Relaxed algorithms should give better complexity and performance. For parallelism, better partitioning schemes for the map-reduce pattern within Weierstrass preparation should be considered. Finally, for the Hensel pipeline, more analysis is needed

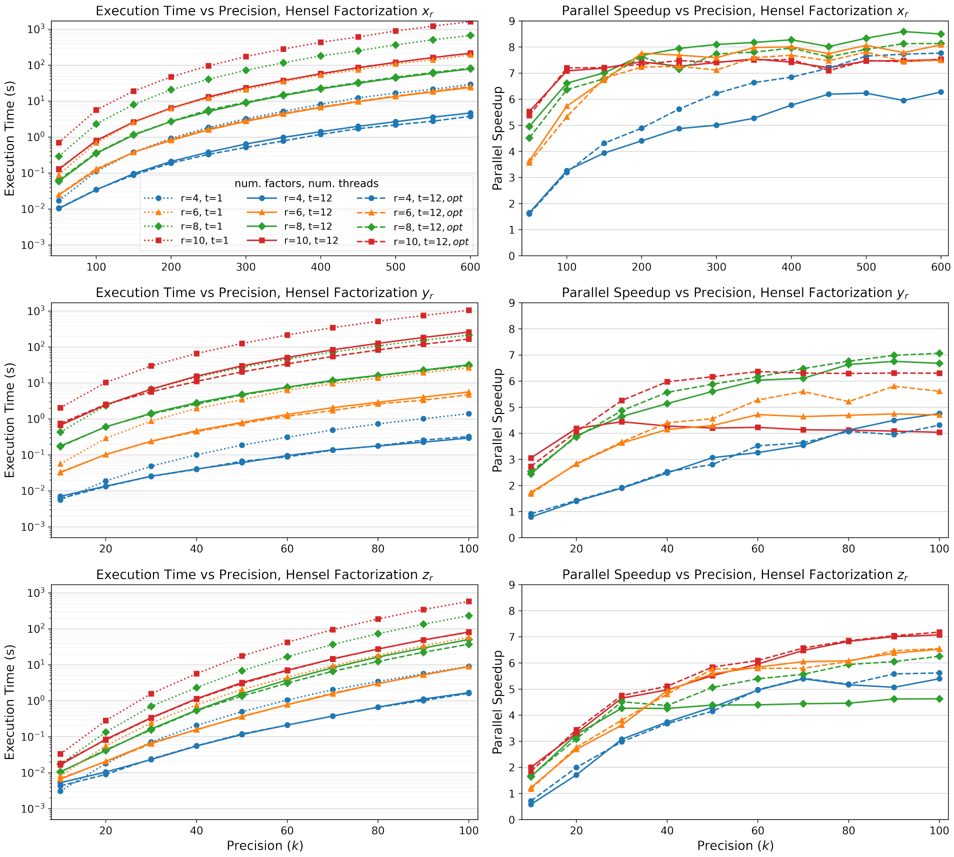

Figure 3: Comparing parallel Hensel factorization for x_r, y_r, and z_r for $r \in \{4, 6, 8, 10\}$. First column: execution time; second column: parallel speedup. For number of threads $t = 12$ resource distribution is determined by Algorithm 9; for $t = 12, opt$ serial execution time replaces complexity measures as work estimates in Algorithm 9, Lines 4–6.

to optimize the scheduling and resource distribution, particularly considering coefficient sizes and the multivariate case.

Acknowledgements

The authors would like to thank NSERC of Canada (award CGSD3-535362-2019).

References

[1] Parisa Alvandi, Masoud Ataei, Mahsa Kazemi, and Marc Moreno Maza. On the extended Hensel construction and its application to the computation of real limit points. *J. Symb. Comput.*, 98:120–162, 2020.

Table 1: Times for updating each factor within the Hensel pipeline, where f_i is the factor with i as the root of $\bar{f}_i$, for various numbers of threads per stage. Complexity-estimated threads use complexity estimates to estimate work within Algorithm 9; time-estimated threads use the serial execution time to estimate work and distribute threads.

		factor	serial time (s)	shift time (s)	Complexity-Est. threads	parallel time (s)	wait time (s)	Time-est. threads	parallel time (s)	wait time (s)
x_4	$k = 600$	f_1	18.1989	0.0012	6	4.5380	0.0000	7	3.5941	0.0000
		f_2	6.6681	0.0666	4	4.5566	0.8530	3	3.6105	0.6163
		f_3	3.4335	0.0274	1	4.5748	1.0855	0	-	-
		f_4	0.0009	0.0009	1	4.5750	4.5707	2	3.6257	1.4170
totals			28.3014	0.0961	12	4.5750	6.5092	12	3.6257	2.0333
y_4	$k = 100$	f_1	0.4216	0.0003	3	0.1846	0.0000	4	0.1819	0.0000
		f_2	0.5122	0.0427	5	0.2759	0.0003	4	0.3080	0.0001
		f_3	0.4586	0.0315	3	0.2842	0.0183	0	-	-
		f_4	0.0049	0.0048	1	0.2844	0.2780	4	0.3144	0.0154
totals			1.3973	0.0793	12	0.2844	0.2963	12	0.3144	0.0155
z_4	$k = 100$	f_1	5.2455	0.0018	6	1.5263	0.0000	7	1.3376	0.0000
		f_2	2.5414	0.0300	4	1.5865	0.2061	3	1.4854	0.0005
		f_3	1.2525	0.0151	1	1.6504	0.1893	0	-	-
		f_4	0.0018	0.0018	1	1.6506	1.6473	2	1.5208	0.7155
totals			9.0412	0.0487	12	1.6506	2.0427	12	1.5208	0.7160

[2] M. Asadi, A. Brandt, C. Chen, S. Covanov, M. Kazemi, F. Mansouri, D. Mohajerani, R. H. C. Moir, M. Moreno Maza, D. Talaashrafi, Linxiao Wang, Ning Xie, and Yuzhen Xie. Basic Polynomial Algebra Subprograms (BPAS) (version 1.791), 2021. `http://www.bpaslib.org`.

[3] Mohammadali Asadi, Alexander Brandt, Mahsa Kazemi, Marc Moreno Maza, and Eric Postma. Multivariate power series in Maple. In *Proc. of MC 2020*, 2021. *(to appear)*.

[4] Mohammadali Asadi, Alexander Brandt, Robert H. C. Moir, and Marc Moreno Maza. Algorithms and data structures for sparse polynomial arithmetic. *Mathematics*, 7(5):441, 2019.

[5] Mohammadali Asadi, Alexander Brandt, Robert H. C. Moir, Marc Moreno Maza, and Yuzhen Xie. On the parallelization of triangular decompositions. In *Proc. of ISSAC 2020*, pages 22–29. ACM, 2020.

[6] Mohammadali Asadi, Alexander Brandt, Robert H. C. Moir, Marc Moreno Maza, and Yuzhen Xie. Parallelization of triangular decompositions: techniques and implementation. *J. Symb. Comput.*, 2021. *(to appear)*.

[7] Jérémy Berthomieu, Grégoire Lecerf, and Guillaume Quintin. Polynomial root finding over local rings and application to error correcting codes. *Appl. Algebra Eng. Commun. Comput.*, 24(6):413–443, 2013.

[8] Alexander Brandt, Mahsa Kazemi, and Marc Moreno Maza. Power series arithmetic with the BPAS library. In *Proc. of CASC 2020*, volume 12291 of *LNCS*, pages 108–128. Springer, 2020.

[9] William H Burge and Stephen M Watt. Infinite structures in Scratchpad II. In *Proc. of EUROCAL 1987*, volume 379 of *LNCS*, pages 138–148. Springer, 1987.

[10] David V Chudnovsky and Gregory V Chudnovsky. On expansion of algebraic functions in power and Puiseux series I. *Journal of Complexity*, 2(4):271–294, 1986.

[11] G. Fischer. *Plane Algebraic Curves.* AMS, 2001.

[12] T. Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library (version 6.1.2)*, 2020. `http://gmplib.org`.

[13] Azzam Haidar, Jakub Kurzak, and Piotr Luszczek. An improved parallel singular value algorithm and its implementation for multicore hardware. In *Proc. of SC 2013.* ACM, 2013.

[14] Maki Iwami. Analytic factorization of the multivariate polynomial. In *Proc. of CASC 2003*, pages 213–225, 2003.

[15] H. T. Kung and Joseph F. Traub. All algebraic functions can be computed fast. *J. ACM*, 25(2):245–260, 1978.

[16] Maplesoft, a division of Waterloo Maple Inc. Maple 2020. `www.maplesoft.com/`.

[17] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation.* Elsevier, 2012.

[18] Panagiotis D. Michailidis and Konstantinos G. Margaritis. Parallel direct methods for solving the system of linear equations with pipelining on a multicore using openmp. *J. Comput. Appl. Math.*, 236(3):326–341, 2011.

[19] Michael B. Monagan and Paul Vrbik. Lazy and forgetful polynomial arithmetic and applications. In *Proc. of CASC 2009*, volume 5743 of *LNCS*, pages 226–239. Springer, 2009.

[20] Vincent Neiger, Johan Rosenkilde, and Éric Schost. Fast computation of the roots of polynomials over the ring of power series. In *Proc. of ISSAC 2017*, pages 349–356. ACM, 2017.

[21] T. Sasaki and F. Kako. Solving multivariate algebraic equation by Hensel construction. *Japan J. Indust. and Appl. Math.*, 16(2):257–285, 1999.

[22] Tateaki Sasaki and Daiju Inaba. Enhancing the extended Hensel construction by using Gröbner bases. In *Proc. of CASC 2016*, volume 9890 of *LNCS*, pages 457–472. Springer, 2016.

[23] Michael L. Scott. *Programming Language Pragmatics (3. ed.).* Academic Press, 2009.

[24] Victor Shoup et al. NTL: A library for doing number theory (version 11.4.3), 2020. `www.shoup.net/ntl/`.

[25] The Sage Developers. *SageMath, the Sage Mathematics Software System (version 9.1)*, 2020. `https://www.sagemath.org`.

[26] Joris van der Hoeven. Relax, but don't be too lazy. *J. Symb. Comput.*, 34(6):479–542, 2002.

[27] Joris van der Hoeven. Faster relaxed multiplication. In *Proc. of ISSAC 2014*, pages 405–412. ACM, 2014.

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