Sparse multivariate Hensel lifting: A high-performance design and implementation.

Michael Monagan[1] and Baris Tuncer[1]

Department of Mathematics, Simon Fraser University, Vancouver, Canada
mmonagan@sfu.ca and tuncer@sfu.ca

Abstract. Our goal is to develop a high-performance code for factoring a multivariate polynomial in n variables with integer coefficients which is polynomial time in the sparse case and efficient in the dense case. Maple, Magma, Macsyma, Singular and Mathematica all implement Wang's multivariate Hensel lifting, which, for sparse polynomials, can be exponential in n. Wang's algorithm is also highly sequential.
In this work we reorganize multivariate Hensel lifting to facilitate a high-performance parallel implementation. We identify multivariate polynomial evaluation and bivariate Hensel lifting as two core components. We have also developed a library of algorithms for polynomial arithmetic which allow us to assign each core an independent task with all the memory it needs in advance so that memory management is eliminated and all important operations operate on dense arrays of 64 bit integers. We have implemented our algorithm and library using Cilk C for the case of two monic factors. We discuss details of the implementation and present experimental timing results.

Keywords: Hensel Lifting, Polynomial Factorization, Cilk C

1 Introduction

Let $a = fg$ where f and g are two irreducible polynomials in $\mathbb{Z}[x_1, x_2, \ldots, x_n]$. Let $\alpha := (\alpha_2, \alpha_3, \ldots, \alpha_n) \in \mathbb{Z}^{n-1}$ be an evaluation point. For a given polynomial $h \in \mathbb{Z}[x_1, x_2, \ldots, x_n]$ let us use the notation $h_j = h(x_1, \ldots, x_j, \alpha_{j+1}, \ldots, \alpha_n)$ so that $a_1 = a(x_1, \alpha_2, \ldots, \alpha_n)$. To factor a we first factor the image a_1 over $\mathbb{Z}$. With high probability $f(x_1, \alpha)$ and $g(x_1, \alpha)$ will be irreducible so we obtain f_1 and g_1. Next we use a process known as Multivariate Hensel Lifting (MHL) to recover f and g from a, f_1, g_1. Maple, Magma, Macsyma, Singular and Mathematica all implement Wang's MHL from [7, 8]. A complete description of Wang's MHL may be found in Ch 6 of Geddes et. al. [3].

The input to Wang's MHL is a, α, f_1, g_1 and a lifting prime p. The evaluation point α and prime p must satisfy $\gcd(f_1, g_1) = 1$ in $\mathbb{Z}_p[x_1]$. The algorithm lifts the factors f_1, g_1 to f_2, g_2 then f_2, g_2 to f_3, g_3 etc. until we obtain f_n, g_n. At the jth step we have $a_j - f_j g_j \mod p = 0$. At the end of this iteration we have $a - f_n g_n \mod p = 0$. Thus for sufficiently large p we obtain the factorization $a = fg$ over $\mathbb{Z}$. The reason Hensel lifting is done modulo a prime p is to avoid an expression swell that would otherwise occur over $\mathbb{Q}$.

Throughout the paper we restrict our presentation to two factors f and g both monic in x_1. We refer the reader to [3] for how to modify MHL for the non-monic and multi-factor cases. Algorithm 1 below shows the jth step of MHL.

Algorithm 1 j^{th} step of Multivariate Hensel Lifting for $j > 1$: Monic Case.

Input : p, $\alpha_j \in \mathbb{Z}_p$, $a_j \in \mathbb{Z}_p[x_1, \ldots, x_j]$, $f_{j-1}, g_{j-1} \in \mathbb{Z}_p[x_1, \ldots, x_{j-1}]$ where a_j, f_{j-1}, g_{j-1} are monic in x_1 and $a_j(x_j = \alpha_j) = f_{j-1}g_{j-1}$.
Output : $f_j, g_j \in \mathbb{Z}_p[x_1, \ldots, x_j]$ such that $a_j = f_j g_j$ or FAIL.

1: $f_j \leftarrow f_{j-1}$; $g_j \leftarrow g_{j-1}$.
2: $\mathit{error} \leftarrow a_j - f_j\, g_j$.
3: **for** $i = 1, 2, 3, \ldots$ **while** $\deg(f_j, x_j) + \deg(g_j, x_j) < \deg(a_j, x_j)$ **do**
4: $\quad c_i \leftarrow$ Taylor coefficient of $(x_j - \alpha_j)^i$ of *error*
5: $\quad$**if** $c_i \neq 0$ **then**
6: $\quad\quad$Solve the MDP $\sigma_i g_{j-1} + \tau_i f_{j-1} = c_i$ in $\mathbb{Z}_p[x_1, \ldots, x_{j-1}]$ for σ_i and τ_i.
7: $\quad\quad$$(f_j,\, g_j) \leftarrow (f_j + \sigma_i \times (x_j - \alpha_j)^i,\, g_j + \tau_i \times (x_j - \alpha_j)^i)$
8: $\quad\quad$$\mathit{error} \leftarrow a_j - f_j\, g_j$
9: $\quad$**end if**
10: **end for**
11: **if** $\mathit{error} = 0$ **then** return f_j, g_j **else** return FAIL **end if**

There are two main computations in Algorithm 1, namely, the multivariate polynomial diophantine equation (MDP) in Step 6, which typically dominates the cost, and the multivariate multiplication of $f_j \times g_j$ in Steps 2 and 8. Wang's method for solving an MDP resembles his Hensel lifting. He first solves the univariate polynomial diophantine equation

$$\sigma_{i1}\, g_{j-1}(x_1, \alpha_2, \ldots, \alpha_{j-1}) + \tau_{i1}\, f_{j-1}(x_1, \alpha_2, \ldots, \alpha_{j-1}) = c_i(x_1, \alpha_2, \ldots, \alpha_{j-1})$$

using the Euclidean algorithm then recovers x_2 then x_3 etc. in σ_i and τ_i. For each x_j there is an iteration on the degree of x_j similar to Algorithm 1. This results in a highly serial algorithm which precludes a parallel implementation.

Wang's solution to the MDP is exponential in j when the evaluation points $\alpha_2, \alpha_3, \ldots, \alpha_{j-1}$ are non-zero. This makes the whole Hensel lifting process exponential for sparse f and g. Polynomial time algorithms were developed by Zippel in 1981 [9], Kaltofen in 1985 [5], and Monagan and Tuncer in 2016 [6].

Let us use the notation $\mathrm{supp}(h)$ to denote the set of monomials appearing in the polynomial h. Monagan and Tuncer [6] solved this exponential problem by observing that if α_j in Algorithm 1 is chosen at random from a sufficiently large set then with high probability the monomials in σ_i for $i \geq 1$ will be contained in the monomials in f_{j-1}, that is $\mathrm{supp}(\sigma_i) \subseteq \mathrm{supp}(f_{j-1})$. Similarly, $\mathrm{supp}(\tau_i) \subseteq \mathrm{supp}(g_{j-1})$ with high probability. They interpolate σ_i and τ_i by picking $\beta_2, \ldots, \beta_{j-1}$ at random from $\mathbb{Z}_p$, computing sufficiently many images of $\sigma_{ik} = \sigma_i(x_1, \beta_2^k, \beta_3^k, \ldots, \beta_{j-1}^k)$ and $\tau_{ik} = \tau_i(x_1, \beta_2^k, \ldots, \beta_{j-1}^k)$ for $1 \leq k$ by solving univariate diophantine equations

$$\sigma_{ik}\, g_{j-1}(x_1, \beta_2^k, \ldots, \beta_{j-1}^k) + \tau_{ik}\, f_{j-1}(x_1, \beta_2^k, \ldots, \beta_{j-1}^k) = c_i(x_1, \beta_2^k, \ldots, \beta_{j-1}^k)$$

for σ_{ik} and τ_{ik} in $\mathbb{Z}_p[x_1]$. Equating coefficients we obtain linear systems. The linear systems are Vandermonde systems which can be solved efficiently in quadratic time and linear space – see Zippel [10]. This improves on Kaltofen's solution to the MDP which results in large unstructured linear systems. The second author has installed this new approach in Maple. It will be available in Maple 2019.

2 High performance considerations

Following Bernardin [1] we first reorganize the computation of c_i in Algorithm 1 to avoid recomputing the entire product $f_j \times g_j$. At the ith iteration of the loop we have $f_j = f_{j-1} + \sum_{k=1}^{i-1} \sigma_k (x_j - \alpha_j)^k$ and $g_j = g_{j-1} + \sum_{k=1}^{i-1} \tau_k (x_j - \alpha_j)^k$ and

$$c_i = \text{coeff}(a_j - f_j g_j, (x_j - \alpha_j)^i) = \frac{{a_j}^{(i)}(\alpha_j)}{i!} - \sum_{k=1}^{i-1} \sigma_k \tau_{i-k}$$

where $a^{(i)}$ is the ith derivative of a_j wrt x_j. So we may write the loop in Algorithm 1 as follows.

1: $f_j \leftarrow f_{j-1}$; $g_j \leftarrow g_{j-1}$; $da \leftarrow \deg(a_j, x_j)$; $df \leftarrow 0$; $dg \leftarrow 0$.
2: **for** $i = 1, 2, 3, \ldots$ **while** $df + dg < da$ **do**
3: $a \leftarrow \partial a / \partial x_j$
4: $c_i \leftarrow a(\alpha_j)/i! - \sum_{k=1}^{i-1} \sigma_k \, \tau_{i-k}$
5: Solve the MDP $\sigma_i g_{j-1} + \tau_i f_{j-1} = c_i$ in $\mathbb{Z}_p[x_1, \ldots, x_{j-1}]$ for σ_i and τ_i.
6: **if** $\sigma_i \neq 0$ set $df \leftarrow i$ **end if**
7: **if** $\tau_i \neq 0$ set $dg \leftarrow i$ **end if**
8: **end for**
9: $f_j \leftarrow f_{j-1} + \sum_{k=1}^{df} \sigma_k \times (x_j - \alpha_j)^k$
10: $g_j \leftarrow g_{j-1} + \sum_{k=1}^{dg} \tau_k \times (x_j - \alpha_j)^k$

How can we parallelize this for a multi-core computer? We are using Cilk C (see [2]), a parallel extension of C available with the gcc compiler. Because of the time needed to start a Cilk process, the units of work should be of size at least 10^4 clock cycles, equivalently, at least 10^3 multiplications in $\mathbb{Z}_p$. Also, small units of work must require no memory allocations, otherwise memory management will become a parallel bottleneck. We propose to reduce the multivariate Hensel lifting in $\mathbb{Z}_p[x_1, \ldots, x_j]$ to Hensel lift bivariate images in $\mathbb{Z}_p[x_1, x_j]$. That is we will Hensel lift x_j in

$$a_j(x_1, \beta_2^k, \ldots, \beta_{j-1}^k, x_j), f_{j-1}(x_1, \beta_2^k, \ldots, \beta_{j-1}^k), \text{ and } g_{j-1}(x_1, \beta_2^k, \ldots, \beta_{j-1}^k).$$

Algorithm 2 is our main unit of work. The complexity estimates on the right count arithmetic operations in $\mathbb{Z}_p$. Here $d_1 = \deg(a, x_1)$ and $d_j = \deg(a, x_j)$).

In Algorithm 2 the loop runs to either $df = \deg(f_j, x_j)$ or $dg = \deg(g_j, x_j)$, whichever is greater. Now since $d_f + d_g = d_j$ the most expensive step is the sum of products $\Sigma = \sum_{k=1}^{i-1} \sigma_i(x_1) \, \tau_{k-i}(x_1)$ in Step 5 which costs $\sum_{i=1}^{d_j} O(i d_1^2) \in O(d_j^2 d_1^2)$ in total. This is the same cost as Bernardin obtains in [1] for two factors. To reduce the cost of Step 5 consider evaluating then interpolating x_1 as follows.

Algorithm 2 HenselLift1: Bivariate Hensel Lift of x_j for $j > 1$.

Input: p, $\alpha_j \in \mathbb{Z}_p$, $a \in \mathbb{Z}_p[x_1, x_j]$, $f_0, g_0 \in \mathbb{Z}_p[x_1]$ where a, f_0, g_0 are monic in x_1, $a(x_1, \alpha_j) = f_0\, g_0$, $\gcd(f_0, g_0) = 1$ and $p > \deg(a_j, x_j)$.
Output : $f_j, g_j \in \mathbb{Z}_p[x_1, x_j]$ such that $a_j = f_j\, g_j$.

1: $da \leftarrow \deg(a, x_j)$; $df \leftarrow 0$; $dg \leftarrow 0$;
2: Solve $sg_0 + tf_0 = 1$ for $s, t \in \mathbb{Z}_p[x_1]$ using the Euclidean Alg. .... $O(d_1^2)$
3: **for** $i = 1, 2, 3, \ldots$ **while** $df + dg < da$ **do**
4: $a \leftarrow \partial a / \partial x_j$ .... $O(d_1 d_j)$
5: $c_i \leftarrow a(x_1, x_j = \alpha_j)/i! - \sum_{k=1}^{i-1} \sigma_k(x_1)\, \tau_{i-k}(x_1)$ .... $O(d_1 d_j) + O(i d_1^2)$
6: Solve $\sigma_i g_0 + \tau_i f_0 = c_i$ for $\sigma_i, \tau_i \in \mathbb{Z}_p[x_1]$ via
7: $\sigma_i \leftarrow (c_i s)$ rem f_0; $\tau_i \leftarrow (c_i - \sigma_i g_0)$ quo f_0 ...... $O(d_1 \deg(f_0, x_1)) \subset O(d_1^2)$
8: **if** $\sigma_i \neq 0$ **then** $df \leftarrow i$ **end if**
9: **if** $\tau_i \neq 0$ **then** $dg \leftarrow i$ **end if**
10: **end for**
11: We have $f_j = f_0 + \sum_{i=1}^{df} \sigma_i(x_1)(x_j - \alpha_j)^i$ and $g_j = g_0 + \sum_{i=1}^{dg} \tau_i(x_1)(x_j - \alpha_j)^i$.
12: **return** $[f_0, \sigma_1, \ldots, \sigma_{df}]$ and $[g_0, \tau_1, \ldots, \tau_{dg}]$

5 Evaluate $\sigma_{il} \leftarrow \sigma_{i-1}(l)$ and $\tau_{il} \leftarrow \tau_{i-1}(l)$ for $0 \leq l \leq d_1$.
for $l = 0$ to d_1 **do** $c_{il} \leftarrow \sum_{k=1}^{i-1} \sigma_{kl} \times \tau_{(i-k)l}$ **end for**
Interpolate $\Sigma(x_1)$ from values $\{(l, c_{il}) : 0 \leq l \leq d_1\}$.
$c_i \leftarrow a(x_1, x_j = \alpha_j)/i! - \Sigma(x_1)$.

Notice that the values σ_{il} and τ_{il} are reused in subsequent iterations. Using Horner's method for evaluation and Newton interpolation the cost of Step 5 becomes $O(d_1^2) + O(id_1) + O(d_1^2) + O(d_1 d_j)$ and the total cost of Algorithm 2 is now $O(d_1^2 d_j + d_1 d_j^2)$. The only new requirment is that $p > d_1$. Note, if $\deg(f_0, x_1) > \deg(g_0, x_1)$ then one may either interchange f_0 and g_0 or use $\tau_i \leftarrow (c_i t)$ rem g_0 and $\sigma_i \leftarrow (c_i - \tau_i f_0)$ quo g_0 to minimize the cost of Step 7.

2.1 Implementation of HenselLift1

In Algorithm 2 there are univariate operations in $\mathbb{Z}_p[x_1]$ and bivariate operations in $\mathbb{Z}_p[x_1, x_j]$. For a high performance implementation we have designed a library of polynomial arithmetic for $\mathbb{Z}_p[x_1]$ and $\mathbb{Z}_p[x_1, \ldots, x_n]$. The data structure for $\mathbb{Z}_p[x_1]$ is just a dense array of coefficients. For $\mathbb{Z}_p[x_1, \ldots, x_n]$ we use a sparse representation. We encode, e.g., the trivariate polynomial $\sum_{i=1}^{t} a_i M_i(x_1, x_2, x_3)$ as two arrays of integers $A = [a_1|a_2|\ldots|a_t]$ and the monomials $X = [M_1|M_2|\ldots|M_t]$ also stored as an array of integers, that is, each monomial $x_1^i x_2^j x_3^k$ in X is stored as the 64 bit integer $2^{42} i + 2^{21} j + k$. Each subroutine in our library has inputs which are either integers or arrays of integers or arrays of arrays of integers. The arrays may be for inputs, outputs, and, if needed, temporary storage. For example, in Step 7 the multiplications $c_i s$ and $\sigma_i g_0$ are done using the C routine

```
# define LONG long long int // 64 bit signed C integer
int polmul64s( LONG *A, LONG *B, LONG *C, int da, int db, LONG p );
```

Here $da = \deg(a, x)$, $db = \deg(b, x)$, the coefficients of $a(x)$ and $b(x)$ are stored in the arrays A and B. The product $c(x) = a(x)b(x) \bmod p$ is computed in the array C which must be an array of size at least $da + db + 1$.

As a second example, in Step 4 we differentiate $a(x_1, x_j)$ with respect to x_j by calling the routine poldiff64s$(A, X, t, 2, 2, p)$ below. Here $a(x_1, x_j)$ is input in the arrays (A, X) and the routine overwrites (A, X) with the derivative $\partial a/\partial x_j$.

```
int poldiff64s( LONG *A, LONG *X, int t, int n, int j, LONG p ) {
// diff(a,x[j]): a is stored as pair (A,X) with t terms in n variables
// compute result in (A,X) and return the number of terms
```

To implement Algorithm 2 we first coded it by allocating space for the polynomials in Algorithm 2, so including space for $\sigma_1, \ldots, \sigma_{df}$ for example. Then we make all polynomials parameters of HenselLift1 so that Algorithm 2 does not allocate any new memory. This is possible because all polynomials have bounded degree. The resulting code will be called on many inputs in parallel and the temporary space can be reused.

2.2 Reduction from multivariate to bivariate Hensel lifting.

Algorithm 3 describes how we reduce Hensel lifting of x_j in f_{j-1}, g_{j-1} to many bivariate Hensel lifts of x_j. When we implemented Algorithm 3 we tested it on polynomials f and g with $100-8000$ terms in $n = 6-15$ variables of degree 7. We observed that almost all the time was spent evaluating a_j at Y_k in step 8. In the next section we discuss how we implement evaluation and how we parallelized it. Here we point out that if instead of evaluating out $x_2, \ldots, x_{j-1}$ we evaluate out $x_3, \ldots, x_{j-1}$, and thus interpolate the σ_i and τ_i from bivariate images in x_1, x_2, then we we likely reduce the number of evaluations s thus leading to a speedup. We describe what we have implemented using a homomorphism diagram.

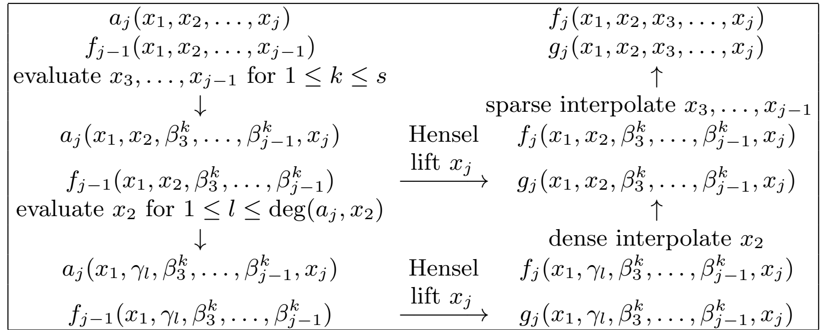

Fig. 1. Homomorphism diagram depicting our evaluation/interpolation strategy

In Figure 1 the reader will see two Hensel lifting steps which represent two possible ways of computing $f_j(x_1, x_2, \beta^k, x_j)$ and $g_j(x_1, x_2, \beta^k, x_j)$. In the first

Algorithm 3 Hensel Lift x_j

Input: Prime p, $\alpha_j \in \mathbb{Z}_p$, Monic polynomials $a_j \in \mathbb{Z}_p[x_1,\ldots,x_j]$ $f_{j-1}, g_{j-1} \in \mathbb{Z}_p[x_1,\ldots,x_{j-1}]$ with $j > 2$, s.t. $a_j(x_1,\ldots,x_{j-1},\alpha_j) = f_{j-1}g_{j-1}$.

1: Let $f_j = \sum_{i=0}^{df} \sigma_i(x_2,...,x_{j-1})x_1^i$ where $\sigma_i = \sum_{k=1}^{s_i} a_{ik}M_{ik}$ where $x_1^i M_{ik}$ are the monomials in $\text{supp}(f_{j-1})$ and $df = \deg(f_{j-1}, x_1)$.
2: Let $g_j = \sum_{i=0}^{dg} \tau_i(x_2,...,x_{j-1})x_1^i$ where $\tau_i = \sum_{k=1}^{t_i} b_{ik}N_{ik}$ where $x_1^i N_{ik}$ are the monomials in $\text{supp}(g_{j-1})$ and $dg = \deg(g_{j-1}, x_1)$.
3: Set $s = \max(s_i, t_i)$.
4: Pick $(\beta_2, \ldots \beta_{j-1}) \in \mathbb{Z}_p$ at random.
5: Compute monomial evaluation sets
 $\{S_i = \{m_{ik} = M_{ik}(\beta_2,\ldots,\beta_{j-1}) : 1 \le k \le s_i\} : 0 \le i \le df\}$ and
 $\{T_i = \{n_{ik} = N_{ik}(\beta_2,\ldots,\beta_{j-1}) : 1 \le k \le t_i\} : 0 \le i \le dg\}$.
 If any $|S_i| \ne s_i$ or any $|T_i| \ne t_i$ try a different choice for $(\beta_2,\ldots,\beta_{j-1})$.
 If this fails **return** FAIL(1). (*p is not big enough*)
6: **for** k from 1 to s **in parallel do** (*Compute univariate images of σ_i and τ_i*)
7: Let $Y_k = (x_2 = \beta_2^k, \ldots, x_{j-1} = \beta_{j-1}^k)$.
8: Evaluate: $a_k, f_0, g_0 \leftarrow a_j(x_1, Y_k, x_j), f_{j-1}(x_1, Y_k), g_{j-1}(x_1, Y_k)$.
9: **if** $\gcd(f_0, g_0) \ne 1$ **return** FAIL(2) *(an unlucky evaluation)*
10: Call HenselLift1$(p, \alpha_j, a_k, f_0, g_0)$ to compute $\sigma_{ik}(x_1)$ and $\tau_{ik}(x_1)$ such that $a_k - f_k g_k = 0$ where $f_k = \sum_{i=0}^{df} \sigma_{ik}(x_j - \alpha_j)^i$ and $g_k = \sum_{i=0}^{dg} \tau_{ik}(x_j - \alpha_j)^i$.
11: **end for**
12: **for** i from 0 to df **do**
13: Construct and solve the $s_i \times s_i$ linear system

$$\left\{ \sum_{k=1}^{s_i} a_{ik}\, m_{ik}^n \;=\; \text{coefficient of } x_1^i \text{ in } \sigma_{in}(x_1) \quad \text{for } 1 \le n \le s_i \right\}$$

for the coefficients a_{ik} of $\sigma_i(x_2,\ldots,x_{j-1})$. Because it is a Vandermonde system in m_{ik} which are distinct by Step 5 it has a unique solution.
14: **end for**
15: Do the same for the $t_i \times t_i$ linear systems to solve for the coefficients b_{ik} of the τ_i.
16: Substitute the solutions for a_{ik} into f_j and b_{ik} into g_j and return(f_j, g_j).

way (the top Hensel lift) the diophantine equations $\sigma_i g_0 + \tau_i f_0 = c_i$ in Step 6 of Algorithm 2 are in $\mathbb{Z}_p[x_1, x_2]$ thus bivariate. One can solve these using dense evaluation and interpolation of x_2 in $O(d_1^2 d_2 + d_1 d_2^2)$ arithmetic operations in $\mathbb{Z}_p$ where $d_2 = \deg(a_j, x_2)$. See Monagan and Tuncer [6].

We coded this approach in Maple as an experiment and found that the most expensive computation is the sum of products $\Sigma = \sum_{k=1}^{i-1} \sigma_k(x_1,x_2)\tau_{i-k}(x_1,x_2)$ in Step 5 of Algorithm 2 which are now bivariate multiplications which cost $O(id_1^2 d_2^2)$. To reduce this cost, we experimented with evaluating and interpolating x_2 which is described by the bottom Hensel lift in Figure 1. So the number of univariate images $f_j(x_1, \gamma_l, \beta^k, x_j)$, $g_j(x_1, \gamma_l, \beta^k, x_j)$ needed to interpolate x_2 is $\max(\deg(f_j, x_2), \deg(g_j, x_2)) < \deg(a_j, x_2) = d_2$. In our current implementation we have parallelized the computation of the Hensel lifts of these images.

2.3 Parallelizing Evaluation

We describe how we parallelize the evaluations in Step 8 of Algorithm 3. Let $a_j = (A, X)$ where the monomials in X are sorted in lexicographical order with $x_1 > x_2 > \ldots > x_j$. We first sort the monomials into $x_1 > x_j > x_2 > \ldots > x_{j-1}$. Now when we evaluate $a_j(x_1, x_j, \beta_2^k, \ldots, \beta_{j-1}^k)$ the evaluated monomials will be sorted on $x_1 > x_j$. Let $a_j = \sum_{i=1}^{t} a_i x_1^{d_i} x_j^{e_i} M_i(x_2, \ldots, x_{j-1})$. Let $A = [a_1, a_2, \ldots, a_t]$ be the array of coefficients, $m_i = M_i(\beta_2, \ldots, \beta_{j-1})$ and $B = [m_1, m_2, \ldots, m_t]$ be the array of monomial evaluations and let Y be the array of monomials $[x_1^{d_1} x_j^{e_1}, \ldots, x_1^{d_t} x_j^{e_j}]$. If we initialize $C_0 := A = [a_1, \ldots, a_t]$ and define $C_k = [a_1 m_1^k, \ldots, a_t m_t^k]$ then we have

$$a_j(x_1, x_j, \beta_2^k, \ldots, \beta_{j-1}^k) = \sum_{i=1}^{t} a_i m_i^k x_1^{d_i} x_2^{e_i} = \sum_{i=1}^{t} Ck_i Y_i$$

and we can compute C_{k+1} from C_k and B using t multiplications with

$$C_{k+1} \leftarrow [B_1 \times C_{k1}, \ldots, B_t \times C_{kt}] = [a_1 m_1^{k+1}, \ldots, a_t m_t^{k+1}]$$

Then we assemble the result from $\sum_{i=1}^{t} C_{k+1,i} Y_i$ which requires adding coefficients of equal monomials in x_1, x_j. Since the monomials in Y_i are already sorted on $x_1 > x_j$ this is $O(t)$. Thus the total number of multiplications needed is st plus those needed to compute $m_1, \ldots, m_t$.

Our first attempt to parallelize this for N cores was to do N evaluations at a time as done by Hu and Monagan in [4]. First compute $C_1, C_2, \ldots, C_N$ and the array $\Gamma = [m_1^N, m_2^N, \ldots, m_t^N]$. To obtain the next N evaluations in parallel, on the kth core compute $C_{k+N} \leftarrow [C_{k1} \times \Gamma_1, \ldots, C_{kt} \times \Gamma_t] = [a_1 m_1^{k+N}, \ldots, a_t m_1^{k+N}]$. One problem with this approach is that we require $\#a$ words of memory for each $C_1, \ldots, C_N$. For one of our benchmark problems where $\#a = 64,000,000$ this is about a half a gigabyte per core. Another problem is that we did not obtain full parallel speedup on our 16 core computer as the computation becomes memory bound when $\#a$ is this large. The following works.

Split $a_j = (A, X)$ into N blocks of size t/N terms. Each core evaluates a block of a_j at β^{k+1} which must be combined later. Numbering the cores $0, 1, \ldots, N-1$ core c computes $C_{kl} \times B_l$ for $c\lfloor t/N \rfloor < l \leq (c+1)\lfloor t/N \rfloor$. We found that we obtained a 20% improvement by also computing the evaluation β^{k+2} at the same time so that we compute two evaluations at a time.

3 Experimental Results

We give two sets of experimental results. The first set (see Table 1) is for polynomials in many variables with relatively low degree. Here, evaluation of a_j is the bottleneck in our method – the time spent Hensel lifting images is negligible. The second set (see Table 2) is for polynomials with higher degree where Hensel lifting becomes the bottleneck. All experiments were performed on a server with

two Intel E5-2660 8 core CPUs running at 2.2GHz (base) and 3.0GHz (turbo) hence the maximum theoretical parallel speedup is a factor of $16.2/3.0 = 11.7$.

In Tables 1 and 2 the factors f and g are of the form $x_1^d + \sum_{i=2}^{t} a_i \prod_{j=1}^{n} x_j^{e_{ji}}$ where the coefficients a_i are chosen randomly from $[1, 999]$ and the exponents e_{ji} randomly from $[0, d-1]$. The timings are for Hensel lifting x_n the last variable only, which is always most of the time. The quantity s in column 4 is the number of images needed to interpolate $x_3, \ldots, x_n$ in Figure 1. Table 1 shows we achieve very good parallel speedup for evaluations. To obtain the parallel speedups for the Hensel lifting in Table 2 we needed to parallelize the evaluations and interpolations of x_2 in Figure 1 as well as the Hensel Lifts.

For Maple we report two timings. The first is the best case of Wang's method where the evaluation points $\alpha_2, \ldots, \alpha_n$ are all 0. To obtain this timing we forced Maple to use x_1 as the main variable (by default, it chooses a variable of least degree) and we added a constant to f and g as Maple requires that the leading and trailing coefficient in x_1 not vanish at α. The second timing is the worst case for Wang's method where all evaluation points are non-zero. It is the actual timing for Maple on these inputs.

				New times (1 core)			New times (16 cores)			Maple 2018	
n	d	t	s	total	(hensel)	(eval)	total	(hensel)	(eval)	best	worst
6	7	500	18	0.098	(0.015)	(0.042)	0.074	(0.019)	(0.008 – 5.2x)	0.411	28.84
6	7	1000	30	0.414	(0.025)	(0.247)	0.180	(0.027)	(0.030 – 8.2x)	1.140	58.46
6	7	2000	47	1.593	(0.041)	(1.132)	0.285	(0.042)	(0.121 – 9.4x)	3.066	99.88
6	7	4000	81	5.072	(0.069)	(4.070)	0.814	(0.074)	(0.380 – 10.7x)	7.173	162.49
6	7	8000	145	12.75	(0.122)	(10.95)	1.896	(0.130)	(0.939 – 11.7x)	15.61	NA
9	7	500	16	0.105	(0.013)	(0.040)	0.101	(0.024)	(0.010 – 4.0x)	1.171	7564.9
9	7	1000	29	0.524	(0.025)	(0.297)	0.233	(0.026)	(0.030 – 11.4x)	3.704	10010.4
9	7	2000	50	2.838	(0.042)	(1.973)	0.483	(0.045)	(0.193 – 10.2x)	13.43	NA
9	7	4000	93	18.35	(0.078)	(14.84)	2.325	(0.083)	(1.350 – 11.0x)	51.77	NA
9	7	8000	164	116.6	(0.139)	(102.5)	11.50	(0.145)	(7.947 – 12.9x)	NA	NA

Table 1: Timings (real time in seconds) for increasing n and t. NA = not attempted.

				New time (1 core)			New time (16 cores)			Maple 2018	
n	d	t	s	total	(hensel)	(eval)	total	(hensel)	(eval)	best	worst
6	10	500	10	0.099	(0.029)	(0.025)	0.081	(0.024 – 1.2x)	(0.006)	0.571	92.49
6	15	500	6	0.134	(0.070)	(0.016)	0.093	(0.034 – 2.1x)	(0.006)	0.751	7956.5
6	20	500	5	0.238	(0.168)	(0.017)	0.130	(0.065 – 2.6x)	(0.005)	0.919	48610.1
6	40	500	3	1.207	(1.128)	(0.015)	0.282	(0.203 – 5.6x)	(0.006)	1.615	NA
6	60	500	3	4.580	(4.486)	(0.015)	0.732	(0.631 – 7.1x)	(0.011)	3.343	NA
6	80	500	3	13.76	(13.65)	(0.016)	1.674	(1.554 – 8.8x)	(0.012)	4.485	NA
6	10	2000	30	1.775	(0.089)	(1.067)	0.374	(0.055 – 1.6x)	(0.121)	5.237	976.94
6	15	2000	18	1.616	(0.221)	(0.706)	0.413	(0.107 – 2.1x)	(0.061)	7.166	23128.5
6	20	2000	12	1.635	(0.451)	(0.480)	0.431	(0.150 – 3.0x)	(0.040)	9.195	NA
6	40	2000	6	4.008	(2.993)	(0.260)	0.854	(0.505 – 5.9x)	(0.038)	15.98	NA
6	60	2000	6	14.25	(13.15)	(0.292)	1.926	(1.500 – 8.8x)	(0.052)	42.32	NA
6	80	2000	4	26.34	(25.25)	(0.217)	3.340	(2.839 – 8.9x)	(0.050)	57.33	NA

Table 2: Timings (real time in seconds) for increasing degree.

4 Implementation notes and Cilk C

We end with some comments about programming in Cilk C. Cilk has a very simple task model. One starts a new task using the `spawn` directive. Typically one creates several tasks in a C for loop inside a C function. One may wait for all the tasks started inside the function to complete using the Cilk `sync;` directive. And that's essentially it! We had few problems with Cilk. But ....

Coding in Cilk C basically means we are coding in C where we must manage the memory needed for every polynomial operation. Naively calling `malloc` and `free` in every subroutine will ruin parallel performance and degrade serial performance. Having to manage memory greatly increases coding effort. To reduce memory allocations we re-designed many algorithms to run in-place, that is, to require no additional memory.

It was very hard work getting an algorithm that took about two days to code in Maple to work in Cilk C. In C there is no array bounds checking. Incorrect memory references result in corrupted data which is difficult to track down. Maintaining an identical version of the code in Maple is helpful here. What we would find helpful is to code in C++ using the array data type, which does not support bounds checking, but have some tool for automatically converting arrays to C++ vectors where array bounds checking is available.

The data structure we use for multivariate polynomials assumes monomials can be packed into a 64 bit integer which limits the degree and number of variables that our software can handle. To accommodate more variables we plan to use the 128 bit integer type available in gcc, thus doubling the number of variables of a given degree that we can handle.

References

1. Bernardin, L.: On Bivariate Hensel Lifting and its Parallelization. In Proceedings of ISSAC 1998, pp. 96–100. ACM Press (1998)
2. Frigo M., Leiserson C.E., and Randall K.H.: The Implementation of the Cilk-5 Multithreaded Language. In Proceedings of PLDI 1998, pp. 212–223, ACM (1998).
3. Geddes K.O., Czapor S.R., Labahn G.: Algorithms for Computer Algebra. Kluwer Academic (1992). ISBN:0-7923-9259-0
4. Hu J., Monagan M.: A fast parallel sparse polynomial GCD algorithm. In Proceedings of ISSAC 2016, pp. 271–278. ACM (2016).
5. Kaltofen E.: Sparse Hensel lifting. In Proceedings of EUROCAL '85, LNCS **204**, 4–17. Springer (1985)
6. Monagan M., Tuncer B.: Using Sparse Interpolation in Hensel Lifting. In Proceedings of CASC 2016, LNCS **9890**, 381–400. Springer (2016).
7. Wang, P.S., Rothschild, L.P.: Factoring multivariate polynomials over the integers. Mathematics of Computation **29**(131), 935–950 (1975)
8. Wang, P.S.: An improved Multivariate Polynomial Factoring Algorithm. Mathematics of Computation **32**(144), 1215–1231 (1978)
9. Zippel, R.E.: Newton's iteration and the sparse Hensel algorithm. In Proceedings of SYMSAC '81, pp. 68–72. ACM (1981).
10. Zippel, R.E.: Interpolating polynomials from their values. J. Symbolic Computation **9**(3), 375–403 (1990)