Parallel sparse interpolation using small primes

Mohamed Khochtali
U.S. Naval Academy
Annapolis, Maryland, U.S.A.
moujahed003@gmail.com

Daniel S. Roche
U.S. Naval Academy
Annapolis, Maryland, U.S.A.
roche@usna.edu

Xisen Tian
U.S. Naval Academy
Annapolis, Maryland, U.S.A.
xisentian@gmail.com

ABSTRACT

To interpolate a supersparse polynomial with integer coefficients, two alternative approaches are the Prony-based "big prime" technique, which acts over a single large finite field, or the more recently-proposed "small primes" technique, which reduces the unknown sparse polynomial to many low-degree dense polynomials. While the latter technique has not yet reached the same theoretical efficiency as Prony-based methods, it has an obvious potential for parallelization. We present a heuristic "small primes" interpolation algorithm and report on a low-level C implementation using FLINT and MPI.

Categories and Subject Descriptors

F.2.1 [**Analysis of algorithms and problem complexity**]: Numerical algorithms and problems—*Computations on polynomials*; G.3 [**Probability and statistics**]: Probabilistic algorithms; G.4 [**Mathematical software**]: Parallel and vector implementations; I.1.2 [**Symbolic and algebraic manipulation**]: Algorithms

Keywords

Sparse interpolation, sparse polynomial, parallel symbolic computation

1. INTRODUCTION

Given a way to evaluate or *sample* an unknown function or procedure, interpolation is the fundamental and important problem of recovering a formula which accurately and completely describes that unknown function. As discovering an arbitrary unknown function from a finite set of evaluations with any reliability would be impossible, some constraints on the size and form of the output are inevitably required.

Here we consider the problem of *sparse polynomial interpolation*, in which we are guaranteed that the unknown function is a multivariate polynomial with bounded degree. Sparse interpolation algorithms date to the 18th century, but have been the focus of considerable recent work in numeric and symbolic computation, with applications ranging from power consumption in medical devices, to reducing intermediate expression swell in mathematical computations [8, 18, 19, 7].

Specifically, this work focuses on algorithms to interpolate an unknown *supersparse* polynomial with integer coefficients, which make efficient use of modern *parallel* computing hardware. Focusing on the "supersparse" (a.k.a. "lacunary") case means that our running time will be in terms of the number of variables, number of nonzero terms, and the logarithm of the output degree.

The first algorithms to solve this problem in polynomial-time were based on the exponential sums technique of Prony, and can efficiently solve the integer problem by working in a large finite field modulo a single "big prime." A number of theoretical improvements and practical implementations have been done in this vein, including work on fast parallel implementations [14, 13].

We consider another type of approach, whereby the unknown sparse integer polynomial is reduced in degree modulo many small primes. This technique, first used by Garg and Schost to avoid the need for discrete logarithm computations in arbitrary finite fields, can also be applied to the integer polynomial case. While it cannot yet match the theoretical efficiency of the big primes algorithms, we will show that the "small primes" method is very effectively parallelized. Furthermore, we develop a practical heuristic version of this method which reduces further the size and number of primes required based on experimental results rather than on the theoretical worst-case bounds.

1.1 Related work

While it would be impossible to list all of the related work on sparse interpolation, we will mention some of the most recent results which are most closely connected to the current study, and which may provide the reader with useful background.

The now-classical approach to sparse interpolation is variously credited to Prony, Blahut [6], or Ben-Or and Tiwari [5]. Only the last of these considered explicitly the case of integer coefficients, but all share the key property of requiring the minimal number $O(T)$ of evaluations in order to recover a polynomial with T nonzero terms.

We refer to these approaches as "big prime" techniques, as the more modern variants [17, 18] adapt to the case of integer coefficients by choosing carefully a single large modulus, then perform the interpolation over a finite field in order to avoid exponential growth in the bit-length of evaluations.

PASCO '15, July 10 - 12, 2015, Bath, United Kingdom
ACM ISBN 978-1-4503-3599-7/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2790282.2790290

The approach which we take in this work is based on that of Garg and Schost [9], who developed the first polynomial-time supersparse interpolation algorithm over an arbitrary finite field. By reducing the unknown polynomial modulo $(z^p - 1)$, the full coefficients are discovered immediately, but the exponents are only discovered after repeating for multiple values of p. This "small primes" approach, described in more details below, has the considerable advantage of relying only on low-degree, dense polynomial arithmetic.

There are, of course, other sparse interpolation methods which do not fit nicely into this big/small prime characterization. Notably, Zippel's algorithm and hybrid variants of it [21, 16], the symbolic-numeric method of [20], and the Newton-Hensel lifting approach of [4].

We point out some recent work on efficient implementations which are of particular interest to the current study. In [14], a new variant of the big prime approach is developed which can be performed variable by variable, in parallel. More recently, [13] investigated a number of tricks and techniques towards practical, efficient sparse interpolation, and posed some new benchmark problems. Their methods are also based on the big prime approach; to our knowledge there has been no reported implementation work on the small primes technique other than the numerical interpolation code reported in [10].

1.2 Our contributions

Suppose $f \in \mathbb{Z}[x_1, x_2, \dots x_n]$ is an unknown polynomial in n variables, with partial degrees less than D in each variable, at most T nonzero terms, and coefficients less than H in absolute value. Then f can be written as

$$f = \sum_{i=1}^{T} c_i x_1^{e_{i1}} x_2^{e_{i2}} \cdots x_n^{e_{in}},$$

where each exponent $e_{ij} < D$ and each $|c_i| < H$.

Given a way to evaluate $f(\theta_1, \dots, \theta_n) \bmod q$, for any modulus $q \in \mathbb{Z}$ and n-tuple of evaluation points $(\theta_1, \dots, \theta_n) \in (\mathbb{Z}/q\mathbb{Z})^n$, the *sparse integer polynomial interpolation problem* is to determine the coefficients c_i and exponent tuples $(e_{i1}, \dots, e_{in})$, for each $1 \le i \le T$.

We will actually consider a slight relaxation of this problem, wherein evaluations are of the form

$$(q, p, d_1, \dots, d_n) \mapsto f(z^{d_1}, \dots, z^{d_n}) \bmod (z^p - 1) \in (\mathbb{Z}/q\mathbb{Z})[z].$$

That is, the coefficients are reduced modulo q and the exponents are reduced modulo p. This is possible without affecting the overall complexity whenever the unknown polynomial f is given as a straight-line program or algebraic circuit, or if the prime q is chosen so that $\mathbb{Z}/q\mathbb{Z}$ has a pth root of unity.

An algorithm for this problem is said to handle the *supersparse* case if it requires a number of evaluations and running time which is polynomial in n, T, $\log D$, and $\log H$. This corresponds to the size of the sparse representation of f as a list of coefficient-exponent tuples, which requires $O(nT \log D + T \log H)$ bits in memory.

We present a randomized algorithm for the sparse integer polynomial interpolation problem, derived from the existing literature, whose running time[*] for provable correctness is

$$\widetilde{O}\left(\left(1 + \tfrac{1}{m} n \log D\right) nT \log D (\log D + \log H)\right), \quad (1)$$

where $m \ge 1$ is the number of parallel processors available for the task. Furthermore, we demonstrate a heuristic variant on this algorithm, which works well in our experimental testing, and reduces the running time further to

$$\widetilde{O}\left(n \log D \log H + \left(1 + \tfrac{1}{m} n \log D\right) T \log H\right) \quad (2)$$

in the typical case that n and $\log D$ are both $O(T \log H)$.

We have implemented this heuristic approach using the C library FLINT for dense polynomial arithmetic and MPI for parallelization. Our experiments demonstrate the smallest effective settings for the parameters in our heuristic approach. With those parameters, we show that the heuristic method is competitive with the state of the art in the single-processor setting, and that its running time scales well with increasing numbers of parallel processors.

Specifically, our contributions are:

1. A sparse interpolation algorithm whose potential parallel speedup is $O(n \log D)$, compared with the $O(n)$ parallel speedup that has been shown in previous work for other sparse interpolation algorithms.

2. A heuristic variant of our algorithm, which is demonstrated to be effective on our (limited) random experiments, that brings the running time complexity of the small primes approach to be competitive with that of the big prime approach.

3. An efficient C implementation of our interpolation algorithm which demonstrates its competitiveness on a standard benchmark problem.

1.3 Organization of the paper

We first outline the two main classes of existing approaches to supersparse integer polynomial interpolation, which we call the "big prime" and the "small primes" methods, in Section 2. We then present our own heuristic method in Section 3, which is based on previously-known "small primes" algorithms, but goes beyond theoretical worst-case bounds on the sizes needed in order to further improve the efficiency.

Section 4 reports the details of our parallel implementation of this heuristic method, and Section 5 presents the preliminary experimental results which demonstrate the efficacy of this approach. Finally, we state some conclusions and directions for further investigation in Section 6.

2. EXISTING ALGORITHMS FOR SUPERSPARSE INTERPOLATION

2.1 "Big prime" methods

The original sparse interpolation algorithm of [5] used evaluations at powers of the first n prime numbers along with Prony's method to deterministically recover the nonzero integer coefficients and exponents of an unknown polynomial. This cannot be considered a "supersparse" algorithm, as it performs the evaluations over the integers directly and requires working with integers with more than D bits.

However, it was soon recognized that, by choosing a single large prime $p \ge D^n$, with $p - 1$ smooth[†] so as to facilitate

[*] Here and throughout we use the *soft-oh* notation in order to simplify the stated running times: a running time is said to be $\widetilde{O}(\phi)$ for some function ϕ if and only if it is $O(\phi (\log \phi)^{O(1)})$.

[†] An integer is said to be *smooth* if it has only small prime factors.

discrete logarithms, a supersparse integer polynomial could be interpolated efficiently modulo p [15, 18]. The basic steps of this approach are as follows:

1. Choose prime p so that $(p-1)$ has a large "smooth" factor greater than D^n and let θ be a primitive element modulo p.
2. For $i = 0, 1, 2, \ldots, 2T-1$, evaluate
$$v_i = f(\theta^i, \theta^{Di}, \ldots, \theta^{D^{n-1}i}) \bmod p.$$
3. Compute the minimum polynomial $\Gamma \in \mathbb{Z}_p[z]$ of the sequence $(v_i)_{i \geq 0}$ with the Berlekamp-Massey algorithm.
4. Factor Γ over $\mathbb{Z}_p[z]$; each root of Γ can be written as $\theta^{e_1+e_2D+\cdots+e_nD^{n-1}}$, corresponding to a single term $cx_1^{e_{i1}} \cdots x_n^{e_{in}}$ of f.
5. Compute T discrete logarithms of the roots, and then the D-adic expansion of each one, to discover the actual exponents $(e_{i1}, \ldots, e_{in})$, for $1 \leq i \leq T$.
6. Once the exponents are known, the coefficients can be computed from the evaluations v_i by solving a transposed Vandermonde system of dimension T.

The two steps which can be trivially parallelized are the evaluations and discrete log computations on steps 2 and 5. However, the dominating cost in the complexity is factoring a polynomial over $\mathbb{Z}_p[x]$ on step 4. This is also confirmed to be the dominating cost in practice by [13], and it is not clear how to efficiently parallelize the factorization. Using the fastest known algorithms, the running time of this step is $\widetilde{O}(T \log^2 p)$ [11], which is at least $\widetilde{O}(n^2 T \log^2 D)$ bit operations from the condition $p > D^n$.

In the description above, the evaluations at powers of θ on step 2 amount to a Kronecker substitution from multivariate to univariate. Of note is the algorithm of [14], which uses a different approach than Kronecker substitution in order to work one variable at a time, gaining a potential n-fold parallel speedup.

2.2 "Small primes" methods

The algorithm described above works the same over an arbitrary finite field, except that the discrete logarithms required in step 5 cannot be performed in polynomial-time in general. This difficulty was first overcome in [9], where the idea is as follows:

1. Choose a series of small primes $p_1, p_2, \ldots$
2. Apply the Kronecker substitution and for each $i = 1, 2, \ldots$ compute $f_i = f(z, z^D, \ldots, z^{D^{n-1}}) \bmod (x^{p_i} - 1)$.
3. Each f_i of maximal sparsity contains all the coefficients of f, and all the exponents modulo p_i. Collect sufficiently many modular images of the exponents in order to recover the full exponents over $\mathbb{Z}$.
4. Recover the multivariate exponents by D-adic expansion of each univariate exponent, and use any f_i of maximal sparsity to discover the coefficient of each term.

There are two significant challenges of this approach. The first is that, in reducing the polynomial modulo $x^{p_i} - 1$, it is possible that some exponents are equivalent modulo p_i, and then multiple terms in the original polynomial will *collide* to form a single term in f_i. By choosing random primes whose values are roughly $\widetilde{O}(T^2 \log D)$, the probability of encountering any collisions can be made arbitrarily small.

The second challenge is how to correlate the exponents from different f_i's in step 3, in order to recover the full exponents via Chinese remaindering. The approach of [9] was to compute an auxiliary polynomial whose roots are the unknown exponents; however this increases the overall running time to $\widetilde{O}(n^2 T^3 \log^2 D \log H)$.

The technique of *diversification* in [10] is another randomization which chooses a random element α and interpolates $f(\alpha z)$ instead of $f(z)$ itself. With high probability, the "diversified" polynomial $f(\alpha z)$ has distinct coefficients, which can then be used to correlate the exponents in different f_i's. This avoids the factoring step and reduces the complexity by a factor of T.

Subsequently, and separately, [1] showed how to allow the magnitude of each p_i to decrease by a factor of T, by allowing some constant fraction of the terms in each f_i to collide. These separate approaches are combined and further improved in [3], which in the typical case that $n \ll \log D \ll \log H$, brings the theoretical complexity down to $\widetilde{O}(nT \log^2 D \log H)$. Observe that this is competitive with the "big primes" algorithm, but could be slower by up to a factor of $\log H$.

The algorithm we describe next first shows how to effectively parallelize this small primes approach, and then further reduces the complexity through a heuristic argument. After the gains from parallelization or from the heuristic improvement, the complexity of our algorithm will be less than the "big primes" approach as well.

3. OUR PARALLEL SMALL PRIMES ALGORITHM

Our algorithm, which is detailed in procedure SparseInterp, is based on the reduction idea in [9], with the diversification method introduced by [10] and the partial collisions handling of [1]. It depends crucially on the parameters k and ℓ, which in theory grow as $O(n \log D)$ and $O(1)$, respectively, but according to our heuristic method can both be treated as constants.

3.1 Algorithm overview

The main idea is first to choose a prime q large enough to recover the coefficients, and then to apply the Kronecker substitution so that we are really interpolating a univariate polynomial $g(z)$ with coefficients in $\mathbb{Z}_q$:

$$g = f(\alpha z, (\alpha z)^D, \ldots, (\alpha z)^{D^{n-1}}) = \sum_{i=1}^{T} c_i z^{E_i} \in \mathbb{Z}_q[z].$$

Each term $c^* x_1^{e_1} \cdots x_n^{e_n}$ of the original polynomial f maps uniquely to a term with exponent $E_i = e_1 + e_2 D + \cdots + e_n D^{n-1}$ and coefficient $c_i = c^* \alpha^{E_i} \bmod q$ in g.

The parameter Q controls the size of the prime q, and so should always be set larger than $2H$ in order to recover the full precision of the coefficients. However, it may be necessary to set Q even larger than this when the height

Procedure SparseInterp(f, n, T, D, H, k, ℓ)

```
Input: Bounds $T, D, H$ for an $n$-variate sparse polynomial $f$ in $n$ variables, with $T$ nonzero terms, partial degrees less than $D$, and integer coefficients less than $H$ in absolute value; and parameters $k, \ell, Q \in \mathbb{Z}_{>0}$.
Output: A list of $t \le T$ coefficients $c_i$ and exponent tuples $(e_{i1}, \dots, e_{in})$. If $k, \ell$ are sufficiently large, these coefficients and exponents comprise the sparse representation of $f$ with high probability.
1  $q \leftarrow$ random prime in the range $[Q, 2Q]$
2  $\alpha \leftarrow$ random element of $\mathbb{Z}_q^*$
3  $(\alpha_0, \dots, \alpha_{n-1}) \leftarrow (\alpha, \alpha^D \bmod q, \dots, \alpha^{D^{n-1}} \bmod q)$
4  $\lambda \leftarrow kT$
5  $\mu \leftarrow \lceil (\ell n \lg D) / \lg \lambda \rceil$
6  $L \leftarrow$ thread-safe list of integer triples
7  for $i = 1, 2, \dots, \mu$ in parallel do
8      $p_i \leftarrow$ random prime in the range $[\lambda, 2\lambda]$
9      $(D_0, \dots, D_{n-1}) \leftarrow (1, D \bmod p_i, \dots, D^{n-1} \bmod p_i)$
10     $f_i \leftarrow f(\alpha_0 z^{D_0}, \dots, \alpha_{n-1} z^{D_{n-1}}) \bmod (z^{p_i} - 1) \in \mathbb{Z}_q[z]$ using dense polynomial arithmetic
11     for $j = 0, 1, \dots, p_i - 1$ do
12         $c_{ij} \leftarrow$ coefficient of $z^j$ in $f_i$
13         if $c_{ij} \ne 0$ then Add $(c_{ij}, j, p_i)$ to $L$
14 Sort $L$ in parallel by the coefficients $c_{ij}$ in each triple
15 $F \leftarrow$ thread-safe list of coefficient/exponent tuples
16 foreach Unique coefficient $c$ in $L$ in parallel do
17     $(d_{c1}, p_{c1}), \dots, (d_{cu}, p_{cu}) \leftarrow$ the $u \ge 1$ exponents and primes appearing with coefficient $c$ in $L$
18     if $u \ge \mu/2$ then
19         $u' \leftarrow$ least integer s.t. $\prod_{1 \le i \le u'} p_{ci} \ge D^n$
20         $E_c \leftarrow$ least integer s.t. $E_c \equiv d_{ci} \bmod p_{ci}$ for $0 \le i \le u'$ via Chinese remaindering
21         $c^* \leftarrow c\alpha^{-E_c} \in \mathbb{Z}_q$, stored as an integer in the range $[-q/2, q/2]$
22         $(e_1, \dots, e_n) \leftarrow D$-adic expansion of $E_c$
23         Add $c^*$ and $(e_1, \dots, e_n)$ to $F$
24 Sort $F$ in parallel by the exponents and return the resulting sparse polynomial
```

bound H is very small. We comment that choosing a random prime q (rather than one with special properties, as in the "big primes" algorithm) is important for the probabilistic analysis below.

The evaluation phase of the algorithm computes the polynomial g modulo $(z^p - 1)$ using dense arithmetic, for many small primes p. The details of how this evaluation is performed will depend on the particular application. In our multivariate multiplication application below, the exponents of the original multiplicands are reduced modulo p, followed by dense polynomial arithmetic in the ring of polynomials modulo $z^p - 1$. More generally, if the unknown polynomial f is given as a straight-line program or arithmetic circuit, each operation in the circuit can be computed over that ring $\mathbb{Z}_q[z]/\langle z^p - 1\rangle$, as in [9]. In our complexity analysis below, we assume a cost of $\widetilde{O}(p \log q)$ for each evaluation on this step, according to the cost of dense degree-p polynomial arithmetic over $\mathbb{Z}_q$.

The next phase of the algorithm is to gather the images of each term, discard those which appear infrequently (and thus were resulting from collisions in the reduction modulo $z^p - 1$), and use Chinese remaindering to recover the exponents of each nonzero term in f.

Observe that the list L can be implemented in any convenient way according to the details of the parallel implementation; it serves only as an unordered accumulator of coefficient-exponent-prime tuples. The output polynomial f could also be considered as an unordered accumulator, followed by another parallel sort before the final return statement.

3.2 Parallel complexity analysis

We state the parameterized running time of the algorithm as follows:

Theorem 1. *Given bounds T, D on the sparsity and degree of an unknown polynomial $f \in \mathbb{Z}[x_1, \dots, x_n]$, and parameters k, ℓ, Q, Algorithm SparseInterp has worst-case running time*

$$\widetilde{O}\Big(\log \ell + n \log D \log Q + kT \log Q + \left(\tfrac{1}{m} \ell n \log D\right)(n + \log D + kT \log Q) \Big).$$

Proof. The computation of μ at the beginning incurs the $\widetilde{O}(\log \ell)$ cost in the complexity, which for our ultimate choices of ℓ will never actually dominate the complexity.

The first loop executes $\lceil \mu/m \rceil \in \widetilde{O}(1 + \frac{1}{m}\ell n \log D)$ times in each thread. Computing the powers of D modulo p_i on Step 9 requires $\widetilde{O}(\log D + n \log p_i)$ bit operations. The subsequent evaluations using dense arithmetic on Step 10 will usually dominate the complexity of the entire algorithm, as each costs $\widetilde{O}(p_i \log q)$, which is $\widetilde{O}(kT \log Q)$. (The addition of this term makes the $\log p_i$ factor in the cost of Step 9 become $\widetilde{O}(1)$.)

Both parallel sorts are on lists of size at most μT, which means their cost of $\widetilde{O}(\frac{1}{m}\ell n T \log D)$ does not dominate the complexity.

The final for loop executes $O(1 + \frac{1}{m}\mu T)$ times in each thread, but the nested if statement can only be triggered $O(T)$ times overall. Within it, the most expensive step is computing each $\alpha^{-E_c} \bmod q$, requiring $\widetilde{O}(n \log D \log Q)$ bit operations. This contributes only $\widetilde{O}(n \log D \log Q)$ to the overall complexity, as the term $\frac{1}{m} T n \log D \log Q$ is already dominated by the parallel cost of Step 10. $\square$

The key feature of procedure SparseInterp is that its potential parallel speedup, from the previous theorem, is a factor of $O(\ell n \log D)$, depending on the number of parallel processors m that are available. This exceeds the $O(n)$ parallel speedup of previous approaches, and means that our algorithm should scale better to a large number of processors when the number of variables and/or degree are sufficiently large.

3.3 Correctness and probability analysis

We first use prior work to prove the bounds necessary to ensure correctness with provably high probability.

THEOREM 2. *If the parameters k, ℓ, Q and bounds D, T, H satisfy*

$$Q \geq \max\left(2H, \tfrac{1}{4}(\ell n T \lg D)^2 D\right),$$
$$k \geq \max(21, \lceil 20n \ln D \rceil), \text{ and}$$
$$\ell \geq 2,$$

then with probability at least 1/2, procedure SparseInterp correctly computes the coefficients and exponents of $f \in \mathbb{Z}[x_1, \ldots, x_n]$.

PROOF. The condition $Q \geq 2H$ guarantees that positive or negative coefficients whose absolute value is at most H will all be distinct modulo q, for any prime $q \geq Q$ as chosen in the algorithm.

Because $\lambda = kT$, Lemma 8 from [1] tells us that, for each p_i, the probability that more than $T/3$ terms collide modulo $z^{p_i} - 1$ is at most 1/4. Since the most number of terms that could collide is T, this means the *expected* number of collisions, for each p_i, is at most $T/2$.

The total number of nonzero coefficients in all polynomials f_i examined on step 13 is at most $\mu T \leq \ell n T \lg D$. Many of these correspond to single terms in f itself, but some will be collisions of terms in f. Using the reasoning of Lemma 4.1 in [2], and the proof of Theorem 3.1 from [10], every coefficient of a single term in f, or a collision that appears in any of the f_i's, is distinct modulo q, due to the bound on Q and because $q \geq Q \geq \frac{1}{2}(\mu T)^2 \lg D$.

We conclude that, with probability at least 1/2, each term in f appears un-collided in at least $\mu/2$ of the polynomials f_i, and furthermore that these coefficients are all distinct and are the only ones that repeat in the list L. This guarantees that the correct coefficients and exponents are recovered in the final loop, and the algorithm outputs the correct interpolated polynomial. □

Applying the bounds in Theorem 2 to the analysis in Theorem 1 gives the provable complexity bound for the algorithm as stated in (1).

As usual, the probability of success in either the provable or the heuristic version of the algorithm (described below) can be increased arbitrarily high by running the same algorithm repeatedly and choosing the most common polynomial returned among all runs to be the most likely candidate for f.

3.4 Heuristic approach

The heuristic version of this procedure is simply to choose appropriate constants for the parameters k and ℓ, with the intuition that there can be some trade-off between the size of each prime (governed by k) and the number of chosen primes (governed by ℓ). Furthermore, the bound on Q required in theory to obtain diversification between the coefficients in f and in any collisions is unnecessarily high for most "typical" polynomials. There do exist pathological counterexamples, but they require the degrees of many terms to be equivalent modulo q. As the prime q is also chosen randomly in our approach, we have a good indication that this heuristic approach will work with high probability for a randomly-chosen sparse polynomial. We state this as a conjecture, which is also backed by the experimental evidence reported in Section 5 below.

CONJECTURE 3. *For any sufficiently large height bound H, and using $Q = 2H$, there exist constants $k, \ell \geq 1$ such that, for a polynomial $f \in \mathbb{Z}[x_1, \ldots, x_n]$ chosen at random with at most T nonzero terms, the probability that algorithm SparseInterp successfully interpolates f is at least 1/2.*

The heuristic complexity under this conjecture is stated in (2).

3.5 Example

Consider an unknown bivariate polynomial $f(x, y)$ with 3 nonzero terms and degree less than 10.

The primes that we are going to use are going to be small for the sake of this example. The primes are 7, 13, and 17.

1. Now we compute f modulo $(z_i^p - 1)$, we get:

$$f \bmod (z^{17} - 1) = 2z^9 + 7z^8 + 3z^3$$
$$f \bmod (z^{13} - 1) = 10z^7 + 2z^4$$
$$f \bmod (z^7 - 1) = 3z^6 + 7z^3 + 2z$$

We notice here that a collision happened with the prime 13. This won't affect our calculation later because resulting coefficient 10 doesn't appear anywhere else (For this example, we are going to assume that we have a good coefficient if it appears twice or more).

Now we use these values to fill the list L. Every triple in L consists of the coefficient, the prime, and the exponent.

coefficient	2	7	3	10	2	3	7	2
exponent	9	8	3	7	4	6	3	1
prime	17	17	17	13	13	7	7	7

We then sort L based on coefficient values.

coefficient	10	7	7	3	3	2	2	2
exponent	7	8	3	3	6	9	4	1
prime	13	17	7	17	7	17	13	7

We use the Chinese remainder theorem to get back the exponent that corresponds to each coefficient.

$$\left.\begin{array}{r} e_1 \bmod 17 = 8 \\ e_1 \bmod 7 = 3 \end{array}\right\} \Rightarrow e_1 = 59$$

$$\left.\begin{array}{r} e_2 \bmod 17 = 3 \\ e_2 \bmod 7 = 6 \end{array}\right\} \Rightarrow e_2 = 20$$

$$\left.\begin{array}{r} e_3 \bmod 17 = 9 \\ e_3 \bmod 13 = 4 \\ e_3 \bmod 7 = 1 \end{array}\right\} \Rightarrow e_3 = 43$$

This results in the univariate polynomial

$$f(z) = 3z^{20} + 2z^{43} + 7z^{59}.$$

Finally, inverting the Kronecker map $f(z) = f(x, y^{10})$, we obtain

$$f(x, y) = 3y^2 + 2x^3y^4 + 7x^9y^5.$$

4. PARALLEL IMPLEMENTATION

We completed a low-level implementation of Procedure SparseInterp written in the C programming language. Our complete source code, as well as the exact source we tested for the comparisons and benchmark problems listed later, is available upon request by email.

We give a few details here on the choices of our implementation, in particular the libraries that were utilized.

4.1 FLINT for sparse and dense polynomial arithmetic

The key advantage to the "small primes" approach which we employed is the reliance on fast subroutines for dense polynomial arithmetic. The experiments we ran always used a word-sized modulus q, and so the most expensive computations involved computing with dense, low-degree polynomials with word-sized modular coefficients.

FLINT (http://flintlib.org/) is a free, open-source C library for fast number theoretic computations [12]. Our dense polynomial arithmetic, which is the dominating cost both in theory and in practice in our experiments, was performed using the `nmod_poly` data type.

In order to store the result of sparse interpolation and complete the correctness testing in our experiments, we also added rudimentary support for sparse integer polynomials on top of FLINT. We created a new data type, `fmpz_sparse`, to represent sparse univariate polynomials in $\mathbb{Z}[x]$ for testing purposes, using FLINT's multiple precision type `fmpz` as both the coefficient and exponent storage for sparse polynomials.

Note that using multiple-precision integers for exponents is especially important, as we have *not* yet completed full multivariate polynomial support within FLINT. Instead, for the purposes of our experiments, we always used a standard Kronecker substitution to store n-variate, degree-D multivariate polynomials as univariate sparse polynomials with degree bounded by D^n. Employing a multiple-precision data type allows for the largest possible degree and number of variables, which is crucial as it is precisely this *supersparse* case in which our approach has the greatest potential parallel speedup.

4.2 MPI for multi-processor parallelism

Message Passing Interface (MPI) allows us to parallelize our algorithm. As stated earlier, since the slowest part of our algorithm involves calculating the unknown polynomial f modulo many polynomials $(x^{p_i} - 1)$, we use MPI to perform each of these evaluations in parallel.

The function `MPI_Init` is called at the beginning of the program to spawn an arbitrary number of processes, as specified on the command line. Each of the allocated processes will be executing separately with separate copies of all variables in the original process. All processes will have unique id numbers. The root process will have id number 0. By knowing processes id's we can separate what each process executes.

We used a master-slave model for our algorithm. The master process evenly distributes how many primes each slave calculates. After getting the primes, the slave process will compute the following for every prime: $\prod_{i=1}^{m} f_i(x) \bmod (x^{p_i} - 1)$, then traverse the resulting univariate polynomial and save all nonzero terms, along with the prime p_i, to an array. This array of triples is sent back to the master process, which later sorts all the concatenated evaluation arrays and uses Chinese remaindering to recover the full polynomial f, as described above.

While our experiments were performed on a multi-core machine, and hence using a simpler threading library would have also worked, our goal in using MPI was to demonstrate the full parallel potential of this approach by explicitly detailing the inter-process communication. Furthermore, the MPI implementation could also be used without modification on heterogeneous clusters or other architectures besides multi-core.

5. EXPERIMENTAL RESULTS

We ran our tests on a machine using an Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz simulating 12 hyper-threaded cores on 6 physical cores, with 32 GB of RAM. We used a Debian GNU system, running the "unstable" branch, with Linux kernel version 3.16.0-4-amd64. This is a bleeding-edge system with the most current versions of all software available within the Debian repositories.

5.1 Determining the parameters k and ℓ

The first task was to determine experimentally what kind of settings for the parameters k and ℓ would be appropriate for our heuristic interpolation method. We found that $k = 38$ and $\ell = 2$ worked for a wide range of problem sizes with almost no failures in the randomized algorithm.

The only exceptions we found here were that when the bound on the number of terms T in the output was very small, even setting $k = 38$ the number of primes p in the range $[\lambda, 2\lambda]$ was simply not sufficient to ensure a high success probability. In these small-size extremes, the value of k was increased to accommodate; in particular, we settled on $k = 50$ and $\ell = 2$ when $T < 1000$, and when $T < 100$ we had to select $k \geq 10000/T$. Under these parameter settings, no failures were observed in any of our experiments.

We comment that a smaller k value and a larger ℓ value would be preferable, because that would increase the number of primes μ and hence the potential parallel speedup for the algorithm, while decreasing the size λ of each prime p_i. However, we found that modestly larger values for ℓ did not allow for k to be reduced and consistently result in correct output. We consider the exploration of some better balance between the parameters k and ℓ as future work.

5.2 Single-threaded performance

The first experiment was to test our algorithm without parallelization against the efficient "big primes" implementation in Mathemagix (http://www.mathemagix.org/), as reported in [13]. We downloaded the source code for the Mathemagix sparse interpolation program, then ran it with $m = 1, 2, 3, 4, 5, 6, 7, 8$ polynomials being multiplied. We kept each polynomial at 20 variables, with degree 40, 3 terms and coefficients up to 2^{30}. Then we ran our algorithm with the same parameters. The results are shown in Taible 1 and summarized in Figure 1. (We also attempted a comparison against the `sinterp` function in Maple 2015, but it was more than an order of magnitude slower in all experiments.)

Our non-parallelized algorithm seems to be on par with the Mathemagix implementation for this range of problem sizes. As a comparison, and to emphasize the main point of our paper, we also show the full parallel speedup for the same problems in Figure 1.

m	variables	terms	max-degree	μ	λ	Mathemagix	Ours (single thread)	Ours (12 threads)
1	20	3	40	14	50 000	0.078	0.035	0.029
2	20	9	80	18	5 000	0.155	0.151	0.048
3	20	27	120	18	6 900	0.305	0.329	0.116
4	20	81	160	18	4 900	0.598	0.323	0.085
5	20	243	200	17	9 900	2.156	0.785	0.175
6	20	729	240	15	39 900	5.053	3.084	0.814
7	20	2187	280	14	80 050	13.333	8.714	2.225
8	20	6561	320	13	321 300	41.070	43.911	10.605

Table 1: Sparse interpolation benchmark times (in seconds)

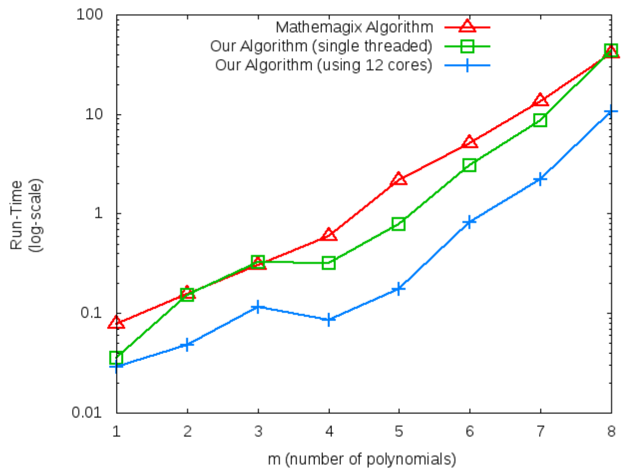

Figure 1: Comparison With Mathemagix Sparse Interpolation Program

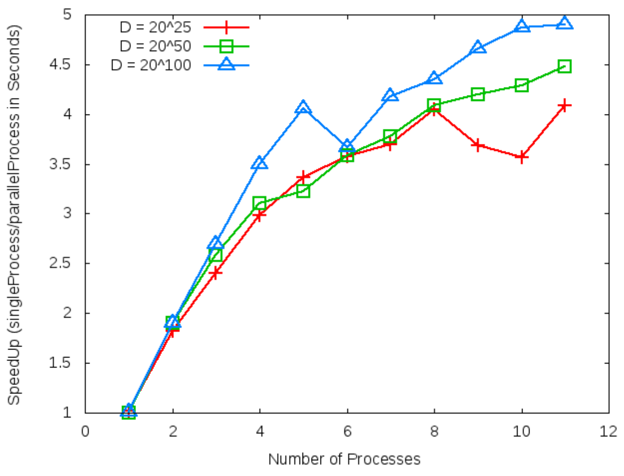

Figure 2: Parallel speedup for our implementation

5.3 Parallel speedup

The second experiment was to test the parallel speedup for our implementation by varying the degree size D and leaving $k = 38$, $l = 2$, partial f_i, and $m = 6$ constant. D was varied from 20^{25}, 20^{50}, 20^{100}. This test was performed 3 separate times and the resultant data was calculated from median of the three trials. The results are seen in Figure 2.

Figure 2 shows a linear speedup increase as the number of parallel processes used gets closer to the physical number of cores on the machine. Additionally, we can see the most significant parallel speedup occurs for the highest degree tested. Recall that on our machine, there are only 6 physical cores that are hyper-threaded to 12 virtual cores. Furthermore, when running only six threads, the "turbo mode" clock rate is increased to 3.8GHz. This may help to explain the dip in performance seen around $m = 6$ parallel processes.

Observe also that the parallel speedup is best, and continues the furthest, in the most extremely sparse case with very high degree.

Our parallel speedup is demonstrated in Figure 2.

6. CONCLUSIONS AND FUTURE WORK

We have shown that the "small primes" sparse interpolation algorithm is competitive with the state of the art, even without parallelization, especially for very sparse problem instances. Furthermore, there is greater potential for parallelism in the small primes technique. These theoretical results are borne out in practice in our experimental results compared to other available software implementations.

There is significantly more work to be done, however, before we might suggest widespread adoption of our heuristic sparse interpolation method. We would like to understand the theory behind the heuristic approach in order to have a less *ad hoc* way of determining the parameters k and ℓ. On the other hand, our implementation could be greatly enhanced with further experimentation on a wider range of benchmark problems and incorporating true multivariate sparse polynomial representations.

7. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under award number #1319994. We thank Andrew Arnold and the anonymous PASCO 2015 reviewers for their comments on an earlier draft of this paper.

8. REFERENCES

[1] A. Arnold, M. Giesbrecht, and D. S. Roche. Faster sparse interpolation of straight-line programs. In V. P. Gerdt, W. Koepf, E. W. Mayr, and E. V. Vorozhtsov, editors, *Proc. Computer Algebra in Scientific Computing (CASC 2013)*, volume 8136 of *Lecture Notes in Computer Science*, pages 61–74. Springer, September 2013.

[2] A. Arnold, M. Giesbrecht, and D. S. Roche. Sparse interpolation over finite fields via low-order roots of

unity. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 27–34, New York, NY, USA, 2014. ACM.

[3] A. Arnold, M. Giesbrecht, and D. S. Roche. Faster sparse multivariate polynomial interpolation of straight-line programs. *CoRR*, abs/1412.4088, 2015.

[4] M. Avendaño, T. Krick, and A. Pacetti. Newton-Hensel interpolation lifting. *Found. Comput. Math.*, 6(1):81–120, 2006.

[5] M. Ben-Or and P. Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 301–309, New York, NY, USA, 1988. ACM.

[6] R. Blahut. Transform techniques for error control codes. *IBM Journal of Research and Development*, 23(3):299–315, May 1979.

[7] B. Boyer, M. T. Comer, and E. L. Kaltofen. Sparse polynomial interpolation by variable shift in the presence of noise and outliers in the evaluations. In *Electr. Proc. Tenth Asian Symposium on Computer Mathematics (ASCM 2012)*, 2012.

[8] A. Cuyt and W. Lee. A new algorithm for sparse interpolation of multivariate polynomials. *Theoretical Computer Science*, 409(2):180–185, 2008. Symbolic-Numerical Computations.

[9] S. Garg and É. Schost. Interpolation of polynomials given by straight-line programs. *Theoretical Computer Science*, 410(27-29):2659–2662, 2009.

[10] M. Giesbrecht and D. S. Roche. Diversification improves interpolation. In *Proceedings of the 36th international symposium on Symbolic and algebraic computation*, ISSAC '11, pages 123–130, New York, NY, USA, 2011. ACM.

[11] B. Grenet, J. van der Hoeven, and G. Lecerf. Randomized root finding over finite FFT-fields using tangent Graeffe transforms. In *Proc. 40th International Symposium on Symbolic and Algebraic Computation*, ISSAC '15, page to appear, 2015.

[12] W. Hart, F. Johansson, and S. Pancratz. FLINT: Fast Library for Number Theory, 2013. Version 2.4.0, http://flintlib.org.

[13] J. van der Hoeven and G. Lecerf. Sparse polynomial interpolation in practice. *ACM Commun. Comput. Algebra*, 48(3/4):187–191, Feb. 2015.

[14] S. M. M. Javadi and M. Monagan. Parallel sparse polynomial interpolation over finite fields. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 160–168, New York, NY, USA, 2010. ACM.

[15] E. Kaltofen, Y. N. Lakshman, and J.-M. Wiley. Modular rational sparse multivariate polynomial interpolation. In *Proceedings of the international symposium on Symbolic and algebraic computation*, ISSAC '90, pages 135–139, New York, NY, USA, 1990. ACM.

[16] E. Kaltofen and W. Lee. Early termination in sparse interpolation algorithms. *Journal of Symbolic Computation*, 36(3-4):365–400, 2003. ISSAC 2002.

[17] E. Kaltofen and L. Yagati. Improved sparse multivariate polynomial interpolation algorithms. In P. Gianni, editor, *Symbolic and Algebraic Computation*, volume 358 of *Lecture Notes in Computer Science*, pages 467–474. Springer Berlin / Heidelberg, 1989.

[18] E. L. Kaltofen. Fifteen years after DSC and WLSS2: What parallel computations I do today [invited lecture at PASCO 2010]. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 10–17, New York, NY, USA, 2010. ACM.

[19] E. L. Kaltofen, W.-s. Lee, and Z. Yang. Fast estimates of hankel matrix condition numbers and numeric sparse interpolation. In *Proceedings of the 2011 International Workshop on Symbolic-Numeric Computation*, SNC '11, pages 130–136, New York, NY, USA, 2011. ACM.

[20] Y. Mansour. Randomized interpolation and approximation of sparse polynomials. *SIAM Journal on Computing*, 24(2):357–368, 1995.

[21] R. Zippel. Interpolating polynomials from their values. *Journal of Symbolic Computation*, 9(3):375–403, 1990. Computational algebraic complexity editorial.