Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

TRINITY COLLEGE DUBLIN

SCHOOL OF MATHEMATICS

Parallel Approaches to Factoring Polynomials over Finite Fields

Samuel McKeown

Supervised by
Nicolas Mascot,
School of Mathematics

A project submitted in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE IN HIGH-PERFORMANCE COMPUTING

Abstract

The Cantor-Zassenhaus algorithm is a widely used method for factoring polynomials over finite fields. We consider a number of modifications to the algorithm to make better use of parallel systems, both in shared-memory and distributed contexts. An implementation is provided and its performance charateristics are investigated. We observe strong scaling in the average case when memory is shared, but gains for distributed systems are modest. Smooth polynomials are less ammenable to the approaches discussed, but some limited speedup can be seen. We find the communication overhead in the implementation to decrease as proportion of runtime as the problem size increases.

This work was supported by RIT (Research IT, Trinity College Dublin).

Contents

1 Introduction

Let $q = p^d$ be a prime power and write $\mathbb{F}_q$ for the finite field with q elements. We will look at the problem of factoring a polynomial $A(x) \in \mathbb{F}_q[x]$ into a product of irreducibles $A(x) = \prod_{i=1}^{k} P_i(x)^{r_i}$. Polynomial factorisation over a finite field is a fundamental operation in computer algebra, being necessary, for example, in the efficient factorisation of polynomials over $\mathbb{Z}$. Another application is as part of the function field sieve method, which calculates discrete logarithms in the context of elliptic curve cryptography.

The Cantor-Zassenhaus algorithm and its variations form one of the major families of factorisation algorithms (the other being those based on Berlekamp's algorithm, as laid out in [7, Section 14.8]). It is used by default by the PARI library [12], and the NTL library [13] recommends it over Berlekamp for most uses.

There have been at least two implementations of the Cantor-Zassenhaus algorithm in parallel, one as part of NTL [11], and the other using the BiPolar library [1]. The approach taken by each was to exploit multi-threading in the underly basic operations, here being multiplication and composition. However, the overhead involved in this low-level parallelism appears to limit scalability, and the need to synchronise after each operation would seem to make this approach unsuitable for distributed computing.

There is also a parallel factorisation algorithm described in [2] and [8], but the emphasis of the authors is on the abstract problem of reducing asymptotic complexity where there are arbitrarily many processors available. They make no claims of its usefulness with a fixed and, compared to the problem size, small number of processors.

The apparent lack of a wider interest in adapting the algorithm for practical parallel systems may be because, in applications such as the ones mentioned above, there tend to many polynomials to be factored at a time. These can be worked on independently by separate threads, and so the question of decomposing the algorithm doesn't arise. The advantage of accelerating the factorisation itself will be in the case of larger polynomials, where the memory requirements of factoring a single polynomial prevents us from working on as many of them as there are threads available.

This report presents some higher-level means of parallelism than are currently in use, and seeks to demonstrate that they work well in practice. Section 2 gives a brief overview of the mathematical background, states the results which underpin the Cantor-Zassenhaus algorithm, and gives a description of the algorithm in its simplest form. Section 3 describes the implementation of the basic polynomial operations and discusses the effectiveness of parallelising these in the straightforward way. Section 4 looks at adjustments which can be made to improve scaling on a shared-memory system, which are extended to distributed systems in section 5.

Scope of the project

The accompanying code[1] implements a handful of approaches for factorisation on shared-memory and distributed systems, and provides test programs demonstrating their use. Threading is managed by OpenMP and inter-node communication in facilitated by MPI. Polynomial arithmetic was implemented from scratch rather than using an existing library, as this seemed to be more time effective than becoming familiar with and re-purposing existing code. The results are checked against PARI's in order to verify correctness.

This implementation is intended primarily as a proof-of-concept, and would be of limited use as a tool for serious computer algebra. Only prime fields are supported, although in principle the adjustments made are applicable to general finite fields. Coefficients are stored always as

[1]Available at: https://try.gitea.io/smckeown/finite_field_factorise

unsigned 32-bit integers, limiting us to $p \leq 2^{32} - 1$ and wasting memory bandwidth when p is small. Though with the execption of a small number of performance-critcal sections, the code is written to be largely independent of the type of the underlying coefficients. It would be possible to address these points by writing a new class for more general coefficients.

Finally, it should be noted that this implementation lacks much of the fine tuning found in existing computer algebra libraries, manifesting in single-threaded performance which is up to an order of magnitude slower than what is currently available.

Note on benchmarks

Unless otherwise noted, the benchmarks quoted were run on an AMD Ryzen *2600* processor at 3.6 GHz, with 6 physical cores and 2-way simultaneous multithreading (SMT). This processor is compatible with AVX2, but must perform the instructions over two cycles and so the corresponding performance uplift is less dramatic than on more recent processors. Programs were compiled with GCC 12.1 and optimisation flags `-O3 -funroll-loops -march=native`.

2 Mathematical background and the Cantor-Zassenhaus algorithm

The correctness of the Cantor-Zassenhaus algorithm relies only on a few elementary facts about polynomials, short enough to be summarised in a small subsection. In fact, the elements of the method were known at least by the 18th Century, but were made into a complete algorithm and popularised by Cantor and Zassenhaus [3]. The historical development is traced in [7, Notes 14.2 and 14.3].

We will recall the definitions and basic results before describing the various steps of the algorithm. In what follows, p is always a prime number and q is always a power of p.

2.1 Finite fields and their polynomials

Prime fields

We denote by $\mathbb{F}_p$ the *integers modulo p*. We identify $\mathbb{F}_p$ with the set $\{0, 1, \ldots, p-1\}$ and define the addition and multiplication by performing the usual integer arithmetic, then taking the remainder by p. For example, we have $3 + 4 = 2 \pmod 5$.

The reason for requiring p to be prime is that this makes $\mathbb{F}_p$ into a *field*, that is, we can take inverses and perform division. We call fields of this form *prime fields*.

Proposition 2.1. *For all $k \in \mathbb{F}_p \setminus \{0\}$, there exists an inverse $k^{-1} \in \mathbb{F}_p$ such that $kk^{-1} = 1 \pmod p$.*

Proof. We have $0 < k < p$, so that $p \nmid k$. Then by the primality of p we must have $\gcd(k, p) = 1$. The extended Euclidean algorithm gives us $a, b \in \mathbb{Z}$ such that

$$ak + bp = 1.$$

Finally, we take the remainders by p to arrive at

$$ak = 1 \pmod p,$$

so that $a = k^{-1} \pmod p$. $\square$

We denote by $\mathbb{F}_p[x]$ the set of *polynomials over* $\mathbb{F}_p$, that is, the set of polynomials with their coefficients in $\mathbb{F}_p$. The usual polynomial operations carry over, but again we take remainders by p. For example, $(4x+1)(3x+1) = 5x^2+1 \pmod 7$.

We say that a polynomial is *monic* if its leading term is equal to 1. It follows from Proposition 2.1 that we can always make a nonzero polynomial monic by dividing by the leading term. For simplicity we'll typically assume that polynomials are monic.

Polynomial division carries over without change to $\mathbb{F}_p[x]$, so that we can take quotients and remainders of polynomials. We say $A(x)$ *divides* $B(x)$ and write $A(x) \mid B(x)$ if the remainder of $B(x)$ by $A(x)$ is zero.

We can define a greatest common divisor of polynomials in the obvious way, once we require it to be monic. The Euclidean algorithm, and the extended version, carry over unchanged, except that we begin by dividing the inputs by their leading terms. We call a pair of polynomials *coprime* if their GCD is 1.

Irreducible polynomials

A monic polynomial $A(x)$ is said to be *irreducible* if there are exactly two (monic) polynomials dividing $A(x)$, which will be $A(x)$ and 1. Said another way, $A(x)$ is irreducible if it is non-constant and has no *non-trivial* factors.

The definition of irreducible polynomials is analogous to that of prime numbers. The following proposition demonstrates some of their similarities.

Proposition 2.2. *Let $A(x), B(x), P(x) \in \mathbb{F}_p[x]$ be polynomials, with $P(x)$ irreducible. If $P(x) \mid A(x)B(x)$, then $P(x) \mid A(x)$ or $P(x) \mid B(x)$.*

Proof. Suppose $P(x) \nmid A(x)$. Since $\gcd(P(x), A(x))$ is a factor of $P(x)$, it must be either $P(x)$ or 1. But it is also a factor of $A(x)$, so by our assumption it must be 1. By using the extended Euclidean algorithm we can find polynomials $C(x), D(x)$ such that

$$A(x)C(x) + P(x)D(x) = 1.$$

Now multiplying by $B(x)$,

$$A(x)B(x)C(x) + B(x)P(x)D(x) = B(x).$$

The left side of this equation is divisible by $P(x)$, and so the right side is too. Hence $P(x)$ divides $B(x)$. $\square$

With the above we can prove, as with integers and prime numbers and by following a similar argument, that every element of $\mathbb{F}_p[x]$ can be written uniquely as a product of irreducible polynomials. If $A(x) = \prod_{i=1}^{k} P_i(x)^{r_i}$ is the product of irreducibles, we refer to $P_i(x)$ as an *irreducible factor* of $A(x)$, and r_i as the *multiplicity* of $P_i(x)$ in $A(x)$. A polynomial is said to be *squarefree* if the multiplicity of each of its irreducible factors is 1. That is, a polynomial is squarefree if it has no repeated factors.

Finite fields

Given $A(x) \in \mathbb{F}_p[x]$ with degree d, we can go further and consider the elements of the quotient ring $\mathbb{F}_p[x]/(A(x))$, which will be the elements $\mathbb{F}_p[x]$ modulo $A(x)$. We identify this with the set of polynomials in $\mathbb{F}_p[x]$ of degree less than d. This is a finite set with p^d elements.

Proceeding as before, we can define operations on $\mathbb{F}_p[x]/(A(x))$ by performing them in $\mathbb{F}_p[x]$ and additionally taking the remainder by $A(x)$. When $A(x)$ is irreducible, the extended Euclidean algorithm can be used as in Proposition 2.1 to find inverses. In this way, $\mathbb{F}_p[x]/(A(x))$ can itself be made into a field. We call $\mathbb{F}_p[x]/(A(x))$ the *finite field* with p^d elements and denote it by $\mathbb{F}_{p^d}$. This is unambiguous because all finite fields with a given number of elements are isomorphic to one another, and further, a finite field with p^d elements can be constructed in this way for any d. These are the only finite fields.

Now with q a prime power, we can consider the polynomials over $\mathbb{F}_q$. The preceding definitions follow through in a totally analogous way.

Example 2.3. Consider $A(x) = x^3 + x^2 + 4 \in \mathbb{F}_5[x]$. It has degree 3, so either it is irreducible, the product of a linear factor and a quadratic factor, or the product of three linear factors. For small p we can quickly find linear factors by testing for roots. We find that

$$f(3) = 27 + 9 + 4 = 40 = 0 \pmod 5,$$

and since $-3 = 2 \pmod 5$, this tells us that $x + 2$ divides $A(x)$. After performing long division we can write $A(x) = (x + 2)(x^2 + 4x + 2)$. The quadratic factor has no roots, and hence we've factored $A(x)$ into irreducibles.

Let $P(x) = x^2 + 4x + 2$ be the irreducible we've just found and consider the finite field $\mathbb{F}_{5^2} = \mathbb{F}_5[x]/(P(x))$. We can try doing arithmetic in $\mathbb{F}_{5^2}$, now that we have a concrete description. For example,

$$(x + 2)(x + 2) = x^2 + 4x + 4 = 2 \pmod{P(x)},$$

and multiplying both sides by $2^{-1} = 3$,

$$(x + 2)(3x + 1) = 1 \pmod{P(x)},$$

so that the inverse of $x + 2$ here is $3x + 1$. More generally, the inverse of any polynomial in $\mathbb{F}_{5^2}$ can be computed using the extended Euclidean algorithm.

It will be useful to note the following application of the Chinese remainder theorem.

Proposition 2.4. *Let $A(x) \in \mathbb{F}_q[x]$ be a polynomial and let $A(x) = \prod_{i=1}^{k} P_i(x)^{r_i}$ be its factorisation into irreducibles. Then we have a corresponding decomposition of the quotient ring*

$$\mathbb{F}_q[x]/(A(x)) \simeq \mathbb{F}_q[x]/(P_1(x)^{r_1}) \times \cdots \times \mathbb{F}_q[x]/(P_k(x)^{r_k}).$$

In particular, if $A(x)$ is squarefree then the quotient ring is a Cartesian product of finite fields.

2.2 Partial factorisations

The factorisation into irreducibles is carried out step-wise, first by grouping factors by multiplicity, then by degree, before finally producing the factors themselves. It will be important to consider each step individually, as the extent and means of parallelism they permit varies dramatically.

Squarefree factorisation

Definition 2.5. Let $A(x) \in \mathbb{F}_q[x]$ be a polynomial with degree d. The *squarefree factorisation* (*SFF*) of $A(x)$ is the product

$$A(x) = \prod_{i=1}^{d} A_i(x)^i,$$

where the $A_i(x)$ are coprime and each is squarefree. Equivalently, $A_i(x)$ is the product of those irreducible factors of $A(x)$ with multiplicity i.

To see how this can be computed, we can look at the more straightforward case of polynomials over $\mathbb{Z}$. If $F(x) \in \mathbb{Z}[x]$ has the squarefree factorisation $F(x) = \prod_{i=1}^{d} F_i(x)^i$, then the product rule gives the derivative as

$$\begin{aligned} F'(x) &= \sum_{i=1}^{d} \left(iF_i'(x)F_i(x)^{i-1} \cdot \prod_{j \neq i} F_j(x)^j \right) \\ &= \sum_{i=1}^{d} iF_i'(x)\frac{F(x)}{F_i(x)}. \end{aligned} \tag{1}$$

The i-th term in the above sum has a factor of $F_i(x)$ of multiplicity $i-1$, and the remaining terms have a factor of multiplicity i. The squarefree factors are coprime, so the multiplicity in the sum is the minimum of these, and

$$\gcd(F_i(x)^i, F'(x)) = F_i(x)^{i-1}.$$

The multiplicative property of the GCD then gives us

$$\gcd(F(x), F'(x)) = \prod_{i=2}^{d} F_i(x)^{i-1}.$$

Performing this operation has the effect of eliminating $F_1(x)$ and decrementing the multiplicities of the remaining factors. Separating out $F_1(x)$ can be done with some additional steps. Let $R(x) = \gcd(F(x), F'(x))$ and let $S(x)$ be the quotient

$$S(x) = \frac{F(x)}{R(x)} = \prod_{i=1}^{d} F_i(x),$$

which is the product of the factors of $F(x)$ without multiplicity. The GCD of $S(x)$ and $R(x)$ will give just the repeated factors, and so finally

$$F_1(x) = \frac{S(x)}{\gcd(S(x), R(x))}.$$

We can repeat this process, now with $R(x)$, to produce the factors of higher multiplicity. The following algorithm avoids some redundant computations but otherwise follows this approach.

Algorithm 2.6 (Squarefree factorisation in $\mathbb{Z}[x]$)**.** Given a polynomial $F(x) \in \mathbb{Z}[x]$, output the squarefree factors $F_1(x), \ldots, F_d(x)$ of $F(x)$.

```
$R(x) \leftarrow \gcd(F(x), F'(x))$        // Product of factors with their multiplicities decremented
$S(x) \leftarrow F(x)/R(x)$                // Product of factors without multiplicity

for $k = 1, \ldots d$ do
    $T(x) \leftarrow \gcd(R(x), S(x))$     // Product of repeated factors, without multiplicity

    $F_k(x) \leftarrow S(x)/T(x)$

    $R(x) \leftarrow R(x)/T(x)$            // Removes non-repeated factors, decrements others
```

```
        $S(x) \leftarrow T(x)$                // Removes found factor. Equivalent to $S(x) \leftarrow S(x)/F_k(x)$
    return $F_1(x), \dots, F_d(x)$
```

The procedure for a polynomial in $\mathbb{F}_q[x]$ is similar, but is complicated by the fact that non-constant polynomials may have zero derivative. Specfically, the terms in Equation 1 with $p \mid i$ will disappear, since they are mutliplied by $i = 0 \pmod{p}$. For example, if we have $A(x) = x^p$, then the derivative is $A'(x) = px^{p-1} = 0$, and their GCD is x^p rather than x^{p-1}.

A modified algorithm which takes account of this possibility is described in [4, Section 3.4.2], but the naive version above is sufficient for a discussion of the parallelism of the computation.

Remark 2.7. The factors found by Algorithm 2.6 will be trivial once $R(x)$ is set to 1. We could use this as an early stopping condition and take the remaining factors to be 1.

In fact, for large q, it's extremely unlikely that the SFF step will require more than one iteration. Call $\mathbb{F}_q^M[x]$ the set of polynomials in $\mathbb{F}_q[x]$ with degree less than M and take M to be large. Let P_k be the proportion of polynomials in $\mathbb{F}_q^M[x]$ which have an irreducible factor of multiplicity at least k. It's easy to show that

$$P_k \leq \frac{1}{q^{k-1} - 1}, \qquad k \geq 2.$$

An informal argument for P_2 goes as follows: let $P(x) \in \mathbb{F}_q[x]$ be irreducible with degree $d \ll M$, and draw $A(x)$ from a uniform distribution on $\mathbb{F}_q^M[x]$. For $P(x)$ to be a repeated factor of $A(x)$, we must have the remainders by $P(x)$ of $A(x)$ and $A(x)/P(x)$ both being zero. These values are independent and have q^d possible, equally likely values, so the probability of this occurring is $1/q^{2d}$. To get an upper bound for $A(x)$ having *some* repeated factor, we can sum over these probabilities for all choices of $P(x)$ and all d. This gives

$$P_2 \leq q \cdot \frac{1}{q^2} + q^2 \cdot \frac{1}{q^4} + \cdots = \sum_{d=1}^{\infty} \frac{1}{q^d} = \frac{1}{q-1}.$$

The exact value of P_k is $1/q^{k-1}$ [10], very close to this bound.

Distinct degree factorisation

Definition 2.8. Let $A(x) \in \mathbb{F}_q[x]$ be a polynomial with degree d. The *distinct degree factorisation (DDF)* of $A(x)$ is the product

$$A(x) = \prod_{i=1}^{d} A_i(x),$$

where $A_i(x)$ is either equal to 1, or has its irreducible factors all of degree i. Equivalently, $A_i(x)$ is the product of those irreducible factors of $A(x)$ with degree i.

We will mostly consider the distinct degree factorisation of a squarefree $A(x)$, so that each of the $A_i(x)$ is also squarefree. In contrast to squarefree factorisation, the method for distinct degree factorisation is particular to finite fields and is quite simple. It relies on the following fact.

Proposition 2.9. *For any $k \geq 1$, the product of the irreducible polynomials in $\mathbb{F}_q[x]$ whose degree divides d is equal to $x^{q^k} - x$.*

A proof is given in [4, Theorem 14.2]. It follows quickly that if $A(x) \in \mathbb{F}_q[x]$ is a squarefree polynomial with distinct degree factorisation $A(x) = \prod A_i(x)$, then

$$\gcd\left(x^{q^k} - x, A(x)\right) = \prod_{i|k} A_i(x). \tag{2}$$

We can find a given $A_k(x)$ by determining $A_i(x)$ for each $i \mid k$, $i < k$, and then dividing them out of $A(x)$ before taking the GCD.

A crucial observation for making this computation feasible is that to calculate the GCD, we don't need to work on the enormous[2] polynomial $x^{q^k} - x$ itself, but only its remainder by $A(x)$.

Algorithm 2.10 (Distinct degree factorisation)**.** Given a squarefree polynomial $A(x) \in \mathbb{F}_q[x]$ of degree d, output its distinct degree factors $A_1(x), \ldots, A_d(x)$.

```
$R(x) \leftarrow A(x)$                          // Stores $A(x)$ less the factors we've found
$B(x) \leftarrow x$

for $k = 1, \ldots d$ do
    $B(x) \leftarrow B(x)^q \pmod{R(x)}$         // $B(x) = x^{q^k} \pmod{R(x)}$

    $A_k(x) \leftarrow \gcd(B(x) - x, R(x))$

    $R(x) \leftarrow R(x)/A_i(x)$                // Remove the factor we found and repeat

return $A_1(x), \ldots, A_d(x)$
```

Remark 2.11. The loop in Algorithm 2.10 need only go up to $k = \lfloor d/2 \rfloor$. If $R(x)$ is non-constant at that point, it must be irreducible, as otherwise it would have a factor of degree less than or equal to $\lfloor d/2 \rfloor$. We can set $A_{\deg R(x)}(x) = R(x)$ and the other factors to 1.

As before, we can also terminate early once $R(x)$ has been set to 1. In contrast to Algorithm 2.6, this will be useful relatively infrequently. The degree of the largest irreducible factor will on average be approximately $0.62d$ [10], meaning that we'll usually reach the $k = \lfloor d/2 \rfloor$ condition first.

Equal degree factorisation

The *equal degree factorisation* (*EDF*) step consists of finding the irreducible factors of a squarefree polynomial whose irreducible factors are known to be of the same degree. We accomplish this by repeatedly splitting such a polynomial into two non-trivial factors ("the Cantor-Zassenhaus split") until we arrive at the irreducibles. The method for $p \geq 3$ relies on the following consequence of Fermat's little theorem.

Proposition 2.12. *Suppose $p \geq 3$ and let $P(x), T(x) \in \mathbb{F}_q[x]$ be polynomials, with $P(x)$ irreducible of degree k and $T(x)$ drawn from a uniform distribution on $\mathbb{F}_q^k[x]$. Set $S(x) = T(x)^{\frac{q^k-1}{2}} - 1$.*

[2]The memory footprint isn't much of a concern here since the calculations could be done with sparse polynomials, but finding x^{q^k} rem $A(x)$ for the first step of the Euclidean algorithm would still require on the order of q^k iterations in the remainder routine.

Then the probability that $S(x)$ is divisible by $P(x)$ is

$$\frac{q-1}{2q} \approx \frac{1}{2}.$$

Further, if $Q(x) \in \mathbb{F}_q[x]$ is another irreducible of degree k, then the divisibility of $S(x)$ by $Q(x)$ is independent of the divisibility by $P(x)$.

We call such an $S(x)$ a *splitting polynomial*. It follows that if $A(x) \in \mathbb{F}_q[x]$ has irreducible factors all of degree k, then $\gcd(S(x), A(x))$ will on average contain half of the factors of $A(x)$. A method to produce splitting polynomials when $p = 2$ is found in [4, Algorithm 3.4.8], but we will only treat the $p \geq 3$ case here.

The ability to produce splitting polynomials leads to an efficient probabilistic EDF algorithm.

Algorithm 2.13 (Cantor-Zassenhaus split)**.** Suppose $p \geq 3$ and that we have a routine $\textsc{rand}(n)$ producing uniformly distributed polynomials of degree less than n.

Given a squarefree polynomial $A(x) \in \mathbb{F}_q[x]$ whose irreducible factors have degree k, output $A(x)$ if it is irreducible or else two non-trivial factors of $A(x)$.

```
if $\deg A(x) = k$ then          // If $A(x)$ has degree $k$ then it must be irreducible by assumption
    return $A(x)$

$B(x) \leftarrow 1$                                           // Stores a factor of $A(x)$

while $B(x) = 1$ or $B(x) = A(x)$ do                // Repeat until $B(x)$ is non-trivial
    $T(x) \leftarrow \textsc{rand}(\deg A(x))$
    $S(x) \leftarrow T(x)^{\frac{q^k-1}{2}} - 1 \pmod{A(x)}$

    $B(x) \leftarrow \gcd(S(x), A(x))$

return $B(x), A(x)/B(x)$
```

The full EDF step involves using Algorithm 2.13 to repeatedly split factors until they're of degree k.

Remark 2.14. The exponent used in producing the splitting polynomial can be rewritten as

$$\frac{q^k-1}{2} = \frac{q-1}{2} \cdot \frac{q^k-1}{q-1} = \frac{q-1}{2} \cdot \sum_{i=0}^{k-1} q^i,$$

so that

$$T(x)^{\frac{q^k-1}{2}} = \left(\prod_{i=0}^{k-1} T(x)^{q^i} \right)^{\frac{q-1}{2}}.$$

The powers in the product can be got by repeatedly taking $T(x)$ to the power of q. This is helpful because it means that we can use only exponents that fit into the same datatype as q. Additionally, we'll see in Section 2.3 that powering by q is equivalent to applying a linear transformation, allowing for the possibility of further optimisations with this approach.

The complete factorisation algorithm comprises of these three steps called in turn.

Algorithm 2.15 (Cantor-Zassenhaus factorisation)**.** Suppose we have routines $\textsc{sff}(A(x))$, $\textsc{ddf}(A(x)))$, and $\textsc{edf}(A(x), k)$ performing the factorisations described above.

Given a polynomial $A(x) \in \mathbb{F}_q[x]$, output the irreducible factors of $A(x)$.

```
$L \leftarrow ()$                                  // Any empty list to store the factors with multiplicity

for $A_i$ in $\textsc{sff}(A(x))$ do                  // Loop through the squarefree free factors
    for $A_{ij}$ in $\textsc{dff}(A_i(x))$ do           // Loop through their distinct degree factors
        $L \leftarrow \big(L, (\textsc{edf}(A_{ij}, j), i)\big)$    // Append to $L$ the factors of $A_{ij}$ and their multiplicity

return $L$
```

Remark 2.16. As discussed in Remark 2.7, for a typical polynomial there will only be one factor in the outer loop not equal to 1. For large q, the probability of a polynomial having no irreducible factors of the same degree is approximately 0.56 [10]. Thus for the majority of polynomials, all of the factorisation is done in a single call to the DDF routine. Even when there are factors of the same degree, they will tend to be small and cheap to split. So the vast majority of the computation in the average case is done by the DDF routine.

2.3 The Frobenius map and the Petr-Berlekamp matrix

Definition 2.17. The *Frobenius map* on $\mathbb{F}_p[x]$ is the map $\phi\colon \mathbb{F}_p[x] \to \mathbb{F}_p[x]$ defined by $\phi(A(x)) = A(x)^p$.

The Frobenius map is clearly multiplicative, but less obviously it is preserves sums and scalar multiplication.

Proposition 2.18. *The Frobenius map is $\mathbb{F}_p$-linear. That is*

$$\phi(kA(x) + B(x)) = k\phi(A(x)) + \phi(B(x)), \qquad \forall k \in \mathbb{F}_p,\ A(x), B(x) \in \mathbb{F}_p[x].$$

In particular, the Frobenius map fixes the elements of $\mathbb{F}_p$.

Proof. That the Frobenius map fixes the elements of $\mathbb{F}_p$ is just Fermat's little theorem. It remains to show that it preserves sums. Applying the binomial theorem and rearranging slightly,

$$\phi(A(x) + B(x)) = A(x)^p + B(x)^p + \sum_{k=1}^{p-1} \binom{p}{k} A(x)^k B(x)^{p-k}.$$

We will be done if we can show that that $p \mid \binom{p}{k}$ for $1 \le k \le p-1$, since this will make the terms of the above sum zero. Expanding the definition gives

$$\binom{p}{k} = \frac{p!}{k!(p-k)!} = \frac{p!}{1 \cdot 2 \cdot \dots \cdot k \cdot 1 \cdot 2 \cdot \dots \cdot (p-k)}.$$

If $1 \le k \le p-1$, then all of the numbers being multiplied in the denominator are strictly less the p and thus not divisible by p. Since p is prime, it follows that the denominator isn't divisible by p. But the numerator is. Hence $p \mid \binom{p}{k}$. $\square$

This property carries over when working modulo some $A(x)$ in the finite dimensional vector space $\mathbb{F}_p[x]/(A(x))$. If $A(x)$ has degree d, we can represent exponentiation by p modulo $A(x)$ as multiplication by a $d \times d$ matrix, called the *Petr-Berlekamp matrix* (over the usual basis). As well as simplifying computations, this matrix can give us knowledge of the factorisation of $A(x)$.

Proposition 2.19. *Let $A(x) \in \mathbb{F}_p[x]$ be squarefree with irreducible factorisation $\prod_{i=1}^{k} P_i(x)$. If Q is the Petr-Berlekamp matrix for $\mathbb{F}_p[x]/(A(x))$, then*

$$\dim \ker(Q - I) = k.$$

That is, the number of solutions to the equation $X^p = X$ in $\mathbb{F}_p[x]/(A(x))$ is equal to p^k.

Recalling the decomposition in Proposition 2.4 and noticing that the Frobenius map fixes these subspaces, the above proposition is equivalent to the statement that on a finite field $\mathbb{F}_q$, the equation

$$X^p = X$$

has exactly p solutions. The solutions are the elements of $\mathbb{F}_p$ embedded in $\mathbb{F}_q$.

This result is stated in [4, Proposition 3.4.9] only for squarefree polynomials, but it remains true even when we drop the assumption.

Theorem 2.20. *Let $A(x) \in \mathbb{F}_p[x]$ be a polynomial with irreducible factorisation $\prod_{i=1}^{k} P_i(x)^{r_i}$. With Q as before, we have*

$$\dim \ker(Q - I) = k.$$

Proof. From the above discussion, it suffices to show that $X^p = X$ has only p solutions in $\mathbb{F}_p[x]/(P(x)^r)$, where $P(x)$ is irreducible and $r \geq 1$. Suppose $A(x)$ is a solution. We would like to show it is constant. Divide it by $P(x)$ with remainder to write

$$A(x) = B(x) + P(x)C(x), \quad \deg B(x) < \deg P(x).$$

Choose k large enough to ensure $p^k > r$. Then after applying the Frobenius map k times, $A(x)$ remains fixed and we get

$$\begin{aligned} A(x) &= B(x)^{p^k} + P(x)^{p^k} C(x)^{p^k} \\ &= B(x)^{p^k} \pmod{P(x)^r}. \end{aligned} \tag{3}$$

Again since $A(x)$ is fixed, we're free to rewrite this as

$$A(x)^{p^{k-1}} = B(x)^{p^k} \pmod{P(x)^r}.$$

This is still true if we reduce modulo $P(x)$, and using the fact that $A(x) = B(x) \pmod{P(x)}$,

$$B(x)^{p^{k-1}} = B(x)^{p^k} \pmod{P(x)}.$$

Hence $B(x) = B(x)^p$ and $B(x)$ is a solution in $\mathbb{F}_p[x]/(P(x))$. But by the proposition, the only solutions here are constants. Thus $B(x)$ is constant, and it follows from Equation 3 that $A(x)$ is too. □

These results still hold when $\mathbb{F}_p$ is replaced by a general finite field $\mathbb{F}_{p^d}$, once we replace the Frobenius map ϕ with its power $\psi = \phi^d$. Given a $A(x) \in \mathbb{F}_q[x]$, we will for convenience refer to the matrix for ψ on $\mathbb{F}_q[x]/(A(x))$ as the Petr-Berlekamp matrix, although this usage may conflict with that of other authors.

```
class Int_mod
{
  public:
    using value_type = uint32_t; // Type for storing values
    using long_type = uint64_t;  // Type for products

  private:
    value_type value;        // Guaranteed to lie in [0, p-1]
    static value_type base; // Stores p

    /* Other static variables for pre-computed constants */

};
```

Listing 1: The data members of the class representing $\mathbb{F}_p$.

3 Implementation of modular and polynomial arithmetic

Having described abstractly the computations we would like to perform, we'll now look at some of the practical considerations of implementing them in a performance-conscious way. The first implementation decision is that of programming language, and a natural choice is C++. As we will see in code snippets, it allows to define types and operators which naturally express the algebraic objects we work with, while still compiling down to an efficient binary.

3.1 Arithmetic in $\mathbb{F}_p$

Representation in memory

The operations on $\mathbb{F}_p[x]$, and thus also $\mathbb{F}_q$, reduce ultimately to arithmetic on the coefficients in $\mathbb{F}_p$, so the choice of how to represent them will have an out-sized influence on overall performance. Listing 1 shows how these elements are represented in the provided implementation.

The product of two variables of an integer type of a given width will in general require a type with twice that width. The largest integer type supported by standard C++ is 64 bits wide, and so 32-bit integers are the largest variables with whose products we can comfortably work. The use of unsigned integers is essential as overflow for signed types is undefined behaviour.

Our code would be cleaner and more flexible if we avoided the use static variables and global state more generally, but the implications on memory usage and the requirements of SIMD instructions make adding additional data to the class impractical. As in Table 1, the storing of p has little observable effect, but the inclusion of any more non-static data members causes unacceptable overhead.

Modular arithmetic

Modular addition and multiplication can easily be done with the built-in modulo operator `%`. The corresponding x86 instruction, DIV, performs an integer division and is known to be relatively expensive, requiring at least ten times the number of cycles as multiplication on the same data types [6].

Optimising compilers can replace this integer division with a combination of cheaper operations when p is known at compile time, and some of the same tricks can be incorporated into our

Class size	Add		Multiply	
	Scalar	AVX2	Scalar	AVX2
32 bits	0.29	0.13	0.30	0.11
64 bits	0.28	0.13	0.30	0.13
92 bits	0.27	0.29	0.30	0.21
128 bits	0.27	0.17	0.30	0.19

Table 1: Comparison of the average time (nanoseconds) taken to add or multiply 32-bit integers over a large number of iterations, when the integers are stored as part of classes of varying size. Auto-vectorisation was turned off for the scalar timings. In each case the compiler can produce code taking advantage of the SIMD instructions, but for the larger classes extra instructions must be used to correctly pack the data into the registers. The 92-bit class doesn't fit evenly into a 256-bit register, so varying alignment between iterations must also be accounted for.

program and done at run time. This is actually necessary if we wish to avail of SIMD support, as none of the available SIMD extensions to either x86 or ARM include an instruction for integer division.

Our addition and multiplication routines are taken essentially unchanged from those described in [9, Functions 1 and 6]. In particular, we use *Barrett reduction* to take remainders more quickly. The authors of that paper consider the problem of vectorising these operations with architecture-specific intrinsic functions, but the advances in auto-vectorisation since have made some of this effort unnecessary.

Table 2 shows the considerable performance improvements over the built-in modulo operator. That there's much tuning that can still be done in this area is suggested by the fact that for fixed p, the compiler can produce multiplication routines which are roughly twice as fast as our efforts. Performance can be improved by making additional assumptions on p, but the cost of choosing the appropriate version dynamically at run time can easily outweigh the benefits. The provided implementation uses routines which are valid for all $p < 2^{32}$.

Scheme	$p = 5$	$p = 7919$	$p = 2^{31} - 1$
Built-in	3.65	3.71	3.67
Improved	0.22	0.21	0.21
Improved (*no O.F.*)	0.14	0.15	0.15
Compile time	0.96	0.84	1.27
Compile time (*no O.F.*)	0.36	0.39	0.45

(a) Modular addition. Addition will never overflow if $p < 2^{31}$, and the simpler schemes for this situation are denoted by (*no O.F.*).

Scheme	$p = 5$	$p = 7919$	$p = 2^{31} - 1$
Built-in	3.77	5.00	7.26
Improved	2.57	2.57	2.53
Improved (*small* p)	1.55	1.52	–
Compile time	0.97	1.16	1.36

(b) Modular multiplication. A faster reduction step can used if we assume $p < 2^{30}$, denoted by (*small* p).

Table 2: Comparison of the average time (nanoseconds) taken to perform modular operations in $\mathbb{F}_p$ over a large number of iterations.

```
class Poly_mod
{
  private:
    ssize_t deg;                   // Equals coeffs.size() - 1
    std::vector<Int_mod> coeffs;   // Stored in ascending degree
};
```

Listing 2: The data members of the class representing $\mathbb{F}_p[x]$.

The last fundamental operation is inversion. This can be accomplished with the extended Euclidean algorithm, as in Proposition 2.1, or with special purpose algorithms such as the ones in [5, Section 11.1.3.a][3]. Another option is to use the identity $a^{-1} = a^{p-2} \pmod p$ from Fermat's little theorem. Table 3 compares the performance these choices. Surprisingly, the Euclidean algorithm performs best despite being the most general.

Scheme	$p = 5$	$p = 7919$	$p = 2^{31} - 1$
Euclid	12	39	81
Fermat	15	113	436
Plus-minus	18	136	294
Prime field	6	57	100

Table 3: Comparison of the average time (nanoseconds) taken to invert an element of $\mathbb{F}_p$ over a large number of iterations.

3.2 Polynomial arithmetic

The class for storing polynomials in $\mathbb{F}_p[x]$ consists just of a standard vector for the coefficients and a variable to store the degree, as shown in Listing 2. A `std::vector` is guaranteed to store its data in contiguous memory, so this doesn't present any problems for vectorisation. The primary run time cost of using `std::vectors` is that memory will always be initialised when allocated, even when the entries are immediately overwritten. We will see later that leaving memory management entirely in the hands of the compiler and standard library comes with measurable overhead.

While the algorithms of Section 2 consist mostly of polynomial remainders, modular exponentiation and GCDs, they are themselves built out of repeated additions and multiplications (with scalar multiplication as a special case of the latter). Thus any lower-level optimisation will necessarily focus on these more basic operations. Other operations are implemented with only minor adjustments from the descriptions in [4].

Addition

The naive way of doing polynomial addition, by looping through the coefficients in order and adding pairwise, is already ideal from the perspective of cache-use and vectorisation. The coefficients are used at most once and in the order they're stored in memory, meaning that they can be reliably pre-fetched. While not available as a standalone instruction, modular addition

[3]The psuedocode for the plus-minus method found in the reference contains small mistakes. Specifically, U should be replaced by U_a and the indentation of lines 5 to 11 should be reduced by one level.

```
Poly_mod a{1, 1};          // a = x + 1
Poly_mod b = a + a + a;    // 1 allocation rather than 3
b = b + std::move(a);      // 0 allocations rather than 1
```

Listing 3: Examples of automatic memory reuse with temporary variables.

can be realised as an integer addition and, conditionally, a subtraction. Each group of modular additions can be mapped to five instructions with the AVX2 extension, and to only four with the AVX-512F extension.

That input coefficients are read at most once means that we can avoid new memory allocations when at least one summand is unneeded afterwards. The addition-assignment operator `+=` can obviously exploit this by overwriting the original values with those of the result, so long as the degree hasn't increased. More generally, C++ has a notion of rvalue references to denote temporary variables whose memory can safely be repurposed. By overloading the `+` and `+=` operators for rvalue references, we can provide the compiler with the means to automatically reuse memory when appropriate. Listing 3 shows some trivial examples of where allocation overhead can be reduced in this way.

Multiplication

Given polynomials

$$A(x) = \sum_{i=0}^{d} a_i x^i,\ B(x) = \sum_{j=0}^{e} b_j x^j,$$

taking the product $A(x)B(x)$ using the classical method will require $(d+1)(e+1)$ multiplications of the coefficients. There are so-called "fast" algorithms with lower asymptotic complexity, and the development of such algorithms is an area of ongoing research. An approach relying on the fast Fourier transform is described in detail in [7, Section 8.3]. We will however heed the advice of the authors of that book, who caution that implementing such algorithms is a delicate and time-consuming task, and restrict ourselves to the usual long multiplication.

By definition, the product is given by

$$A(x)B(x) = \sum_{k=0}^{d+e} c_k x^k, \quad \text{where } c_k = \sum_{i+j=k} a_i b_j.$$

Using this formula to compute each coefficient leaves us with a large number of independent calculations which may be trivially parallelised. However, when we find explicit limits for the latter sum, we get

$$c_k = \sum_{j=\max(0,k-e)}^{\min(k,d)} a_{k-j} b_j.$$

The number of entries in the sum varies with k, and we have to access the coefficients of one of the multiplicands in reverse order. While such a loop can benefit from SIMD instructions, cycles must be wasted to account for the variable loop length and to permute coefficients. We find better performance by rearranging the formula to be

$$A(x)B(x) = \sum_{i=0}^{e} b_j A(x) x^i. \tag{4}$$

```
// 1: Using the definition for the coefficients
for (ssize_t k = 0; k <= prod_deg; ++k)
    for (ssize_t j = max(0, k - max_deg); j <= min(k, min_deg); ++j)
        result.coeff(k) += max_poly.coeff(k - j) * min_poly.coeff(j);

// 2: Rearranging for better SIMD and memory access
for (ssize_t j = 0; j <= min_deg; ++j)
    for (ssize_t i = 0; i <= max_deg; ++i)
        result.coeff(i + j) += max_poly.coeff(i) * min_poly.coeff(j);

// 3: Explicitly writing it as scalar mult. and shifts
for (ssize_t j = 0; j <= min_deg; ++j)
    result.shift_add(max_poly * min_poly.coeff(j), j);
```

Listing 4: Different options for arranging the multiplication loop. Here `shift_add()` is a function for combining a left-shift and addition-assignment.

That is, we consider the product as the sum of scalar multiples and shifts of $A(x)$. With this change, the inner loop has a fixed number of iterations and the memory access is much more predictable.

As well as being a pleasing simplification, the reduction to scalar multiplication presents a further opportunity for optimisation. Following the idea of [9, Function 8], we can compute a "pre-inverse" for the scalar at the start of the routine. This is a value which is then used to perform more efficiently the reduction of the integer products modulo p. When using this approach to polynomial multiplication, we should prefer to take the coefficients of the smaller polynomial as the scalars, so as to minimise the number of pre-inverses required.

Explicit vectorisation

The majority of the runtime will be spent in the scalar multiplication routine, so it will be fruitful to attempt to improve it further. A peculiar trait of the SIMD instructions for integer multiplication is that they only act on half of the values in a register, since the products will in general require twice the space to store. For this reason, although a 256-bit vector register can hold eight 32-bit integers, auto-vectorisation will tend to pack only four at a time. In our case the computation also includes reduction modulo p, which can be done on all of the values in a vector register simultaneously. Current compilers don't make optimal use of SIMD instructions here, and there is room for hand-tuning.

The accompanying implementation includes versions of the scalar multiplication routine using AVX2 and SSE4.1 intrinsic functions[4]. Although such a function isn't given explicitly in [9], one can be written relatively painlessly by rearranging elements of the examples they provide. The improvement comes by filling the vector register with 32-bit integers and performing the multiplication one half at a time. Also, it's possible to combine the halves of the computation sooner if we know that $p < 2^{31}$. Unlike with the modular arithmetic, the cost of checking the size of p at each call to the routine isn't noticeable, so this tweak is incorporated into the implementation.

The three loops in Listing 4 and the hand-vectorised versions are compared in Table 4. While

[4]The routine is called `scalar_mult_simd()` and is defined in `Poly_mod/Poly_mod_mult.cc`. Using AVX-512 instrinsics would allow for a simplified function, but they remain widely unsupported.

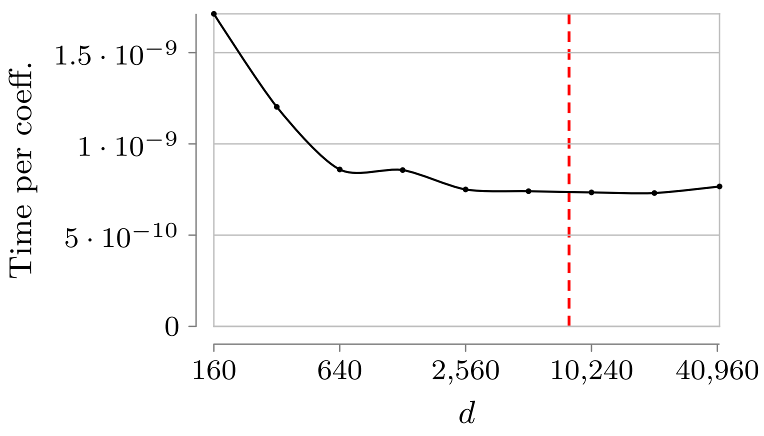


Figure 1: Time per coefficient multiplication as the degree of the polynomials increase, with $p < 2^{31}$ and using the SIMD multiplication routine. The degree of the largest polynomial which can fit in L1 cache is marked in red.

they all display quadratic scaling, the absolute performance of the optimised routines is many times better. It's also interesting to observe that the more optimised routines require larger polynomials to reach their the maximum efficiency.

Scheme	$d = 640$	$d = 1280$	$d = 2560$	$d = 5120$
Definition	7.89	7.42	7.44	7.41
Reordered	2.90	2.91	2.84	2.86
Scalar Mult.	2.01	1.75	1.68	1.67
SIMD	1.03	0.95	0.91	0.90
SIMD (*small* p)	1.00	0.88	0.79	0.79

Table 4: Comparison of the average time (nanoseconds) per coefficient multiplication in multiplying two polynomials of degree d, with various implementations of the classical algorithm. Multiplying by $(d+1)^2$ will give the time for a complete polynomial multiplication.

Cache optimisation and allocation overhead

We might ask if cache use would be improved by blocking the polynomials, similar to what is done in BLAS libraries. Here we would divide the larger polynomial into pieces small enough to fit comfortably in L1 cache, and work on them in turn. This might be relevant for very large polynomials, but Figure 1 fails to show a visible drop in performance when the polynomials are too large to fit in L1 cache[5]. The polynomials we work with will fit at least into the L2 cache of any recent processor, and it seems that this isn't distant enough to cause a bottleneck.

Another consideration is that, written as in 4, the polynomial multiplication routine is wasteful with memory allocation. In each iteration of the loop it will perform a scalar multiplication and store the result as a temporary object, and each must be freshly allocated and then deallocated shortly afterwards. The multiplication of two degree d polynomials will incur the overhead for memory allocation at least d times. The remainder routine also creates temporary variables at every iteration of its loop.

[5]The L1 data cache of the CPU tested is 32KB and can hold at most 8000 32-bit coefficients.

```
{
    // An empty polynomial large enough to store scalar products
    Poly_mod buffer(max_deg);

    for (ssize_t i = 0; i <= min_deg; ++i)
    {
        max_poly.buffered_mult(min_poly.coeff(i), buffer);
        result.shift_add(buffer, i);
    }
} // buffer deallocated at the end of scope
```

Listing 5: Multiplication with pre-allocated memory. `buffered_mult()` is a function for performing scalar multiplication and storing the result directly to the provided memory. The RAII features of C++ remove the need to explicitly free the memory when we're finished.

By allocating a single buffer at the beginning of a routine and using it to store what would otherwise be temporaries, we can greatly ameliorate this overhead. Listing 5 shows that this can be done without a large loss of readability. Table 5 gives timings for multiplication and remainders with and without the use of buffers, from which we can deduce that allocation adds around 10% of overhead when it is managed totally automatically. Multiplications and remainders would otherwise account for the vast majority of allocations, so similar changes aren't needed elsewhere.

Degree	$p < 2^{31}$		$2^{31} \leq p < 2^{32}$	
	Temporaries	Buffers	Temporaries	Buffers
640	0.88	0.88	1.01	0.97
1280	0.83	0.79	0.99	0.90
2560	0.80	0.75	0.95	0.89
5120	0.81	0.74	0.94	0.89

Table 5: Comparison of the average time (nanoseconds) per coefficient multiplication in multiplying polynomials, using either temporary variables or a dedicated buffer to store intermediary results.

3.3 Low-level parallelism

Implementing the Cantor-Zassenhaus algorithm as described in Section 2 is quite straightforward, once a full set of algebraic operations is available. Table 6 shows a breakdown of the functions called in factoring a random polynomial of moderate degree. It's apparent that the bulk of the runtime is spent computing modular exponents or GCDs, using left-right binary powering[4, Algorithm 1.2.3] and the Euclidean algorithm[4, Algorithm 3.2.1], respectively. Neither of these algorithms can be parallelised in an obvious way, but the underlying additions and multiplications can be.

The use of user-defined reductions introduced in OpenMP 4.0 allows us to concisely distribute the iterations of the loops used for the basic operations, as in Listing 6. Figure 2 shows that there is a speedup for multiplication even with small degree polynomials. Curiously, using buffers to store the scalar multiples has a negative impact when we use more than one thread, so temporary variables are used in this case instead. Addition and scalar multiplication, in contrast, appear to be entirely memory-bound and show no measurable gains.

Routine	Prop. of runtime
SFF	$<$ **0.01**
DDF	1.00
Poly. mod. exp.	0.94
Poly. remainder	0.49
Scalar mult.	**0.36**
Poly add.	**0.13**
Poly. mult.	**0.45**
GCD	0.06
Poly. remainder	0.06
Scalar mult.	**0.03**
Poly add.	**0.02**
Other	**0.01**
EDF	$<$ **0.01**

Table 6: A break down of the time spent in various routines while factoring a random polynomial of degree 1000 with $p = 7919$. A value in bold indicates that it isn't broken down further. Typical of the average case, this particular polynomial had no repeated factors and had only low degree factors to be split in the EDF step.

Less than half of the runtime of the overall algorithm is spent on polynomial multiplication, so Amdahl's law would suggest a speedup of no more than ${}^1\!/\!_{(1-0.45)} \approx 1.8$. Table 7 shows this to be too generous. We see incremental gains which each core added, but in no case is the efficiency close to desirable. We can note that symmetric multi-threading provides no benefit here. Speedup appears to decrease with d but seems independent of p, and the $O(d^3 \log p)$ complexity of the algorithm is apparent.

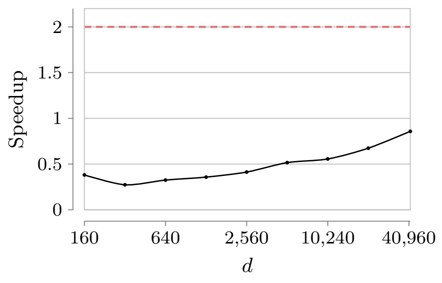
4 Parallelisation for shared-memory systems

If we wish to see reasonable scaling for the Cantor-Zassenhaus algorithm across multiple threads, then Remark 2.16 and Table 6 suggest that this will be largely accomplished in the average case if Algorithm 2.10 can be adapted to do more work in parallel. It is though possible to construct polynomials for which the DDF step isn't the dominant component of runtime. *Smooth* polynomials, whose irreducible factors are all of small degree, are of particular interest and their factorisation will be done primarily in the EDF step. Thus, for the sake of completeness we will discuss what can be done for each step of the algorithm.

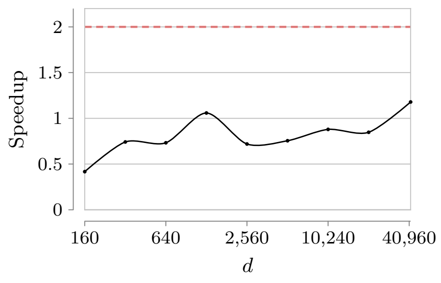
4.1 Pre-computing the Petr-Berlekamp matrix

The most expensive aspect of the DDF step is the modular exponentiation which is performed at every iteration. It isn't clear if this operation can be easily parallelised in general, but we can notice that the exponent q remains fixed. This can be realised as repeated application of the Frobenius map, which we denoted by ψ in Section 2.3. Moreover, this is a linear map whose application is trivially parallelised once we have the corresponding matrix Q.

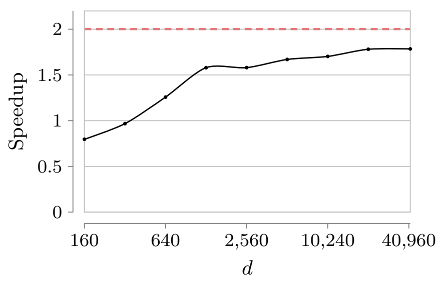
For ease of implementation it is useful to consider Q as a row vector of polynomials, so write $Q_k(x) = \psi(x^k)$ for the k-th column of Q. Then with a polynomial $B(x) = \sum_{j=0}^{e} b_j x^j$ of degree



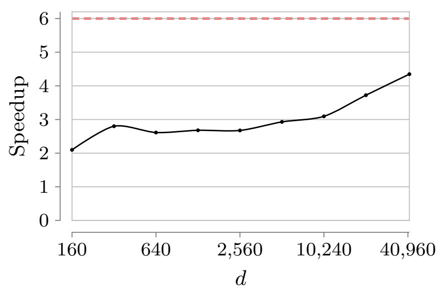
(a) Addition with $N = 2$.



(b) Scalar multiplication with $N = 2$.



(c) Polynomial multiplication with $N = 2$.



(d) Polynomial multiplication with $N = 6$.

Figure 2: Speedup observed when multi-threading the basic operations. The ideal speedup is shown in red.

```
// User-defined reduction
#pragma omp declare reduction(poly_sum:Poly_mod : omp_out += omp_in) \
                    initializer(omp_priv = Poly_mod{0})

// Main loop of the addition routine
#pragma omp parallel for
    for (ssize_t i = 0; i <= min_deg; ++i)
        result.coeff(i) = coeff(i) + a.coeff(i);

// Main loop of the multiplication routine
#pragma omp parallel for reduction(poly_sum : result)
    for (ssize_t j = 0; j <= min_deg; ++j)
        result.shift_add(max_poly * min_poly.coeff(j), j);
```

Listing 6: Using OpenMP to speed up basic operations.

Threads	$d = 1000$			$d = 2000$		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	1.49	1.00	1.00	12.81	1.00	1.00
2	1.31	1.14	0.57	12.40	1.03	0.52
4	1.11	1.35	0.34	10.57	1.21	0.30
6	1.06	1.41	0.24	10.08	1.27	0.21
12 (SMT)	1.18	1.27	0.11	10.58	1.21	0.10

(a) $p = 5$.

Threads	$d = 1000$			$d = 2000$		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	11.39	1.00	1.00	76.41	1.00	1.00
2	10.55	1.08	0.54	71.14	1.07	0.54
4	8.58	1.33	0.33	62.71	1.22	0.30
6	8.37	1.36	0.23	59.67	1.29	0.21
12 (SMT)	8.72	1.30	0.11	60.82	1.26	0.10

(b) $p = 7919$.

Table 7: Comparison of the average time (seconds) taken to factor a random polynomial as we increase the number of threads, using low-level parallelism.

$e < d$, we have

$$B(x)^q = \sum_{j=0}^{e} b_j Q_j(x) \pmod{A(x)}.$$

We can notice that this formula is similar to the one we gave for polynomial multiplication in Equation 4, but without shifts.

Evaluating $\psi(B(x))$ with the more general left-right binary powering method requires

$$\lfloor \log_2 q \rfloor + wt(q)$$

polynomial multiplications and remainders, where $wt(q)$ is the Hamming weight of q. The cost of applying a linear map as above is closer to that of a single multiplication[6]. The lack of remainders with this approach is particularly significant, as this was the component of the computation which we had previously struggled to do in parallel.

The trade-off is that the polynomials $Q_k(x)$ must be pre-computed, and this is an expensive task if done naively by direct evaluation of ψ. The d evaluations required is already more than the $d/2$ evaluations needed in Algorithm 2.10, and further, the polynomials in the latter become progressively smaller as the program proceeds. We can note that although it wouldn't be sensible to do so in serial, pre-computing Q in this way is embarrassingly parallel and so are the subsequent applications of Q, so this could lead to an improvement given enough processors.

A crucial observation in making the pre-computation of Q more worthwhile is that we have the recurrence relation

$$Q_{k+1}(x) = Q_k(x)Q_1(x) \pmod{A(x)}.$$

[6]At least in the absence of fast multiplication algorithms.

We know that $Q_0(x) = 1$, and we can find $Q_1(x)$ by directly evaluating $\psi(x)$. The rest of the columns can then be got with a single multiplication and remainder each. The total number of operations involved becomes

$$\lfloor \log_2 q \rfloor + wt(q) + d - 2. \tag{5}$$

This recurrence relation does create a dependence between calculations and unfortunately means that the task is no longer embarrassingly parallel.

As a compromise between these two schemes, we can calculate directly a starting column for each thread, then apply the recurrence relation to fill out the rest.

Algorithm 4.1 (Pre-computation of Q in parallel)**.** Suppose we have a routine INTERVAL(n) which partitions the indices $\{0, \ldots, n-1\}$ into intervals among the available threads, and returns the endpoints of the calling thread's allocation.

Given a polynomial $A(x) \in \mathbb{F}_q[x]$ of degree d, output the Petr-Berlekamp matrix for $A(x)$.

```
$Q_1(x) \leftarrow \psi(x)$            // The underlying multiplication can benefit from multi-threading

parallel
    $(s, t) \leftarrow$ INTERVAL($d$)

    $Q_s(x) \leftarrow Q_1(x)^s \pmod{A(x)}$            // For large q, this is cheaper than $\psi(x^s)$

    for $i = s+1, \ldots, t$ do            // Loop through this thread's remaining columns
        $Q_i(x) \leftarrow Q_{i-1}(x)Q_1(x) \pmod{A(x)}$    // Fill out the rest with the recurrence relation

return $\left[Q_0(x) \mid \cdots \mid Q_{d-1}(x)\right]$
```

Remark 4.2. With N as the number of threads, we can give a conservative upper bound on the total number of multiplications and remainders as

$$d + N(2\log_2 d - 1) + 2\log_2 q,$$

and on the number of operations per-thread as

$$\frac{d}{N} + 2\log_2 d + 2\log_2 q - 1. \tag{6}$$

Typically we will have d large relative to N and $\log q$, and in that case this bound will be dominated by the linear d term. Hence we should expect the amount of work per thread to be roughly inversely proportional to N.

When $q < s$ in the above, it would be cheaper to find a thread's starting column by calculating $\psi(x^s)$ directly. Checking if this is the case isn't expensive, and gives a valuable improvement when q is small relative to d. The bound on per-thread operations becomes

$$\frac{d}{N} + 4\log_2 q - 1.$$

Remark 4.3. As part of the pre-computation, and elsewhere, will have pairs of operations consisting of a multiplication of polynomials of degree $d-1$, and the remainder of their product by $A(x)$. These have near-identical cost.

More generally, for polynomials $A(x)$, $B(x)$ with degrees d, e and $d \leq e$, their product will require $(d+1)(e+1)$ multiplications and additions of the coefficients, whereas taking the remainder of $B(x)$ will typically require $(d+1)(e-d+1)$ of each. The counts are equal when $e = 2d$.

Remark 4.4. Applying the recurrence relation as above would involve at most $d(2d-1)$ multiplications and additions of the coefficients. But when $q < d$, we simply have $Q_1(x) = x^q$. Multiplication by $Q_1(x)$ can be realised as a left-shift by q degrees and hence requires no arithmetic. The shifted polynomial will have degree at most $d+q-1$, and taking its remainder by $A(x)$ would involve at most $q(d+1)$ coefficient multiplications and additions.

By taking the ratio of operation counts, we can predict that using shifts in evalutating the recurrence relation will affect the cost by a factor of approximately $q/2d$, thus turning the pre-computation from an $O(d^3)$ task into one in $O(qd^2)$. This suggests that the pre-computation can be made extremely cheap (relative to the factorisation) when $q \ll d$, and that using shifts will improve performance so long as $q < 2d$.

With $d = 2000$ and $q = 2011$ and on a single thread, calculating the matrix with multiplications took 11.85s and with shifts took 5.98s, agreeing closely with our prediction.

```
// The matrix should be applied to b mod a
const auto reduced = b % a;

// We can re-use the user-defined reduction
#pragma omp parallel for reduction(poly_sum : result)
    for (ssize_t i = 0; i <= reduced.degree(); ++i)
        result += matrix[i] * reduced.coeff(i);
```

Listing 7: Using OpenMP to speed up matrix application.

Threads	Pre-computation			Multiplication		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	11.92	1.00	1.00	3.74	1.00	1.00
2	6.47	1.84	0.92	1.86	2.01	1.00
4	3.29	3.62	0.91	1.03	3.64	0.91
6	2.34	5.09	0.85	0.89	4.20	0.70
12 (SMT)	2.29	5.20	0.43	0.69	5.39	0.45

Table 8: Comparison of the pre-computation times (seconds) and application times (milliseconds) for the matrix Q, with $d = 2000$ and $p = 7919$. Note that $p > 2d$.

The routine for applying the matrix Q is almost identical to the polynomial multiplication routine, and can similarly take advantage of the optimisations made to scalar multiplication. Listing 7 shows the simple fashion in which matrix application is parallelised.

Timings for pre-computing Q and for using it to find modular exponents are given in Table 8. As could be expected from the estimates above, we approach optimal speedup for the pre-computation when d is large relative to the number of threads. Efficiency for the application of the matrix drops more quickly, which might be explained by the fact that it's a much shorter computation in comparison and additionally requires a reduction.

Updating the matrix

Over the course of Algorithm 2.10, we remove from $A(x)$ the factors we've found and store this reduced polynomial as $R(x)$. Let Q' the matrix for $R(x)$ at a given iteration. We can avoid finding Q' and continue to use Q to perform exponentiation, since $R(x)$ is a factor of $A(x)$, and since it will still be true that

$$Q \cdot B(x) = B(x)^q \pmod{R(x)}.$$

But proceeding this way will leave us with a $B(x)$ which isn't reduced modulo $R(x)$ (we will only have $\deg B(x) < \deg A(x)$ rather than $\deg B(x) < \deg R(x)$). We will end up doing computations on larger polynomials than if we had used Q'.

We can produce Q' by taking the remainders by $R(x)$ of the first $\deg R(x)$ columns of Q, and it's trivial to split this work across threads. The cost of doing such an update isn't completely negligible, and whether or not it's worthwhile depends on the distribution of the factors of $A(x)$. For example, if $A(x)$ has a single linear factor and many quadratic factors, then an update to the matrix after the first iteration would be wasted.

Empirically, it does appear that updating after every iteration is better than not updating at all, with the difference in runtimes lying between 0% and 20%. In the absence of a simple rule to determine the optimal points at which to update the matrix, we use this simple approach.

Effects on scaling

Table 9 shows how the runtime for factoring a particular polynomial breaks down, now using the Petr-Berlekamp matrix. The improved efficiency has the effect of reducing the proportion of the program spent on modular exponentiation. The computation of GCDs now makes up the majority of runtime, and we haven't yet taken steps to parallelise this part. Although a great benefit to absolute performance, proportionally the situation is close to that in Table 6, with less than half of our program effectively parallelised.

The scaling of the algorithm after these changes is shown in Table 10. When p is small the pre-computation takes only a small fraction of the runtime, leaving the majority spent on the serial GDCs. The speedup in this case is remains close to what it was before, but it becomes more acceptable when p is larger and more time is spent on the highly parallel pre-computation.

4.2 Multiple GCDs in parallel

Amdahl's law and Table 9 suggest that the potential speedup for our algorithm will not far exceed 2, so long as the GCDs continue to be calculated in serial. A parallel algorithm for polynomial GCDs does exist [2], but would provide at most an $O(N^{\frac{2}{3}})$ speedup[7].

We can avoid the problem of effectively parallelising the GCD computation by instead examining the dependencies between the GCDs needed by Algorithm 2.10. The order of operations ensures that on the k-th iteration, that the reduced polynomial $R(x)$ has no irreducible factors of degree less than k. This avoids the ambiguity of Equation 2, where in general we pick up all of the factors of degree dividing k, and not just those of degree exactly k. We are though free to rearrange the operations, so long as we are careful in removing these unwanted factors.

Algorithm 4.5 (Distinct degree factorisation in parallel)**.** Let N be the number of available threads. Suppose we have a routine $\textsc{matrix}(A(x))$ which implements Algorithm 4.1, and a routine $\textsc{update}(Q, R(x))$ which performs the update described in the preceding section.

[7]The speedup isn't given explicitly but is implied by their bounds on the number of processors required.

Routine	Prop. of runtime
SFF	$<$ **0.01**
DDF	1.00
Matrix pre-comp.	**0.30**
Poly. mod. exp.	0.09
Matrix mult.	0.09
Scalar mult.	**0.07**
Poly add.	**0.02**
GCD	0.51
Poly. remainder	0.51
Scalar mult.	**0.34**
Poly add.	**0.15**
Other	**0.02**
Matrix update	**0.10**
EDF	$<$ **0.01**

Table 9: A break down of the time spent in various routines while factoring the polynomial of Table 6, which had $d = 1000$ and $p = 7919$, but now using a matrix for exponentiation. Note that when $p \ll d$, the contribution from pre-computation becomes less than 1%. A value in bold indicates that it isn't broken down further.

Given a squarefree polynomial $A(x) \in \mathbb{F}_q[x]$ of degree d, output the distinct degree factors $A_1(x), \ldots, A_d(x)$ of $A(x)$.

```
$R(x) \leftarrow A(x)$                                  // Stores $A(x)$ less the factors we've found
$Q \leftarrow \textsc{matrix}(R(x))$                     // Stores the matrix for $R(x)$

$B(x) \leftarrow x$                                     // Stores the current power of $x$ modulo $R(x)$
$k \leftarrow 1$                                        // The loop counter
while $k \leq d$ do

    // Compute the next $N$ powers of $x$
    for $i = 0, \ldots N-1$ do
        $B(x) \leftarrow Q \cdot B(x)$                  // Multiplication can be done in parallel
        $B_i(x) \leftarrow B(x)$                        // $B_i(x) = x^{q^{k+i}} \pmod{R(x)}$

    // Do $N$ GCDs in parallel
    parallel for $i = 0, \ldots N-1$ do
        $C_i(x) \leftarrow \gcd(B_i(x) - x, R(x))$      // $C_i(x) = \prod_{j|(k+i),\, j \geq k} A_j(x)$

    // Clean up duplicate factors (in serial)
    for $i = 0, \ldots N-1$ do
        for $j = 0, \ldots i-1$ do
            if $j \mid i$ then
                $C_i(x) \leftarrow C_i(x)/C_j(x)$       // The order ensures that now $C_j = A_{k+j}$

    // Set distinct degree factors and remove them from $R(x)$
```

Threads	$d = 1000$			$d = 2000$		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	0.65	1.00	1.00	4.83	1.00	1.00
2	0.58	1.10	0.55	4.12	1.17	0.59
6	0.48	1.36	0.23	3.53	1.37	0.23
12 (SMT)	0.51	1.28	0.11	3.53	1.37	0.11

(a) $p = 5$.

Threads	$d = 1000$			$d = 2000$		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	2.34	1.00	1.00	18.10	1.00	1.00
2	1.55	1.51	0.75	11.66	1.55	0.78
6	0.93	2.53	0.42	6.72	2.69	0.45
12 (SMT)	0.90	2.61	0.22	6.60	2.74	0.23

(b) $p = 7919$.

Table 10: Comparison of the average time (seconds) to factor a random polynomial after the adjustments described in Section 4.1.

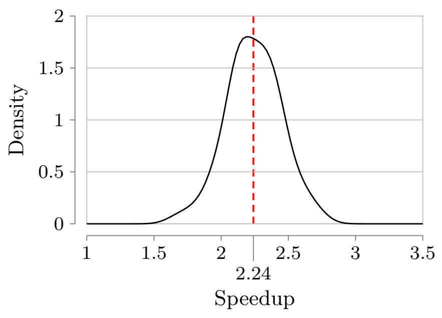
```
        for $i = 0, \ldots N-1$ do
            $A_{k+i}(x) \leftarrow C_i(x)$
            $R(x) \leftarrow R(x)/A_{k+i}(x)$

        $Q \leftarrow$ UPDATE$(Q, R(x))$                    // Updates $Q$ if $R(x)$ was reduced
        $k \leftarrow k + N$                                // We do $N$ degrees per iteration

    return $A_1(x), \ldots, A_d(x)$
```

Remark 4.6. Duplicate factors can only occur when $k+j \mid k+i$ for some distinct thread indices i and j. If $k > N$, then it's always true that $k+j > {}^{k+i}/_2$, and so $k+j$ can't divide $k+i$. Hence the clean-up step is only needed on the first iteration. For the remaining iterations, the only potentially costly work done in serial is the division of $R(x)$ by the factors found.

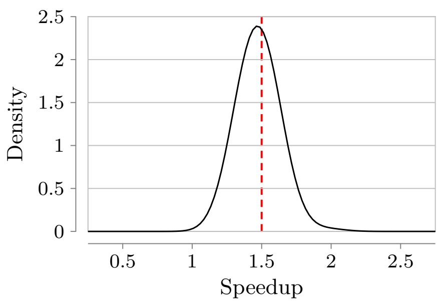
In fact, we can avoid the duplication of factors altogether by restricting the number of GCDs we perform in the early iterations. If on the k-th iteration we compute the GCDs for only $\min(N, 2^{k-1})$ degrees, then by the same reasoning as before we can safely skip the clean-up step. In practice the average cost of removing duplicate factors is negligible and this complication isn't necessary.

Remark 4.7. We essentially do the computation for N degrees between attempts to reduce $R(x)$ by dividing out the factors found. If $A(x)$ has factors of degrees between k and $k+N-1$, then we will have missed opportunities to reduce $R(x)$. This entails doing the exponentiation and GCD calculations with unnecessarily large polynomials, which are both $O((\deg R(x))^2)$ operations.

The difference would be pronounced for polynomials with factors whose degrees are densely clustered. Fortunately, a random $A(x)$ will have on average $\log d$ irreducible factors and the gaps



(a) $p = 5$, $d = 1000$



(b) $p = 7919$, $d = 1000$

Figure 3: Observed distribution of the performance improvement from calculating multiple GCDs in parallel, with $N = 4$. The ratio is taken with the time to factor using only the adjustments described in Section 4.1. The dotted line indicates the average improvement.

between their degrees will tend to increase with d [10], so that the number of missed reductions is proportionally small.

Threads	$d = 1000$			$d = 2000$		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency
2	0.36	1.77	0.89	2.52	1.92	0.96
6	0.17	3.90	0.65	1.18	4.08	0.68
12 (SMT)	0.14	4.50	0.38	0.94	5.16	0.43

(a) $p = 5$.

Threads	$d = 1000$			$d = 2000$		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency
2	1.29	1.82	0.91	9.58	1.89	0.94
6	0.51	4.56	0.76	3.91	4.63	0.77
12 (SMT)	0.47	4.97	0.41	3.42	5.30	0.44

(b) $p = 7919$.

Table 11: Comparison of the average time (seconds) to factor a random polynomial after the adjustments described in Section 4.2. The single-threaded timings are unaffected.

Table 11 demonstrates the effect of doing multiple GCDs at once. We see that scaling in the average case is greatly improved, especially with larger degrees. We no longer see a drastic distinction between small and large p, and symmetric multi-threading now gives measurable gains.

However, more work is being done overall and this isn't always beneficial. By factoring a large product of linear polynomials we can get close to the worst-case "speedup" of $1/N$ in the DDF step. Figure 3 gives the distribution of the improvement for uniformly distributed polynomials, confirming that the approach has a positive effect in the overwhelming majority of cases.

```
const auto split_factors = split(a, k); // Two non-trivial factors

omp_set_nested(true);                          // Enable nesting
omp_set_max_active_levels(log_2(num_threads)); // Limit depth

#pragma omp parallel for num_threads(2)
for (int i = 0; i < 2; ++i)
{
    auto subfactors = edf(split_factors[i], k); // Recursive call

#pragma omp critical
    factors.merge(subfactors); // Combine results
}
```

Listing 8: Nested `parallel` regions in the EDF routine.

4.3 Parallelism in equal degree factorisation

Assume now that $A(x)$ is squarefree and has irreducible factors all of degree k. Let $c = {}^{d}/_{k}$ be the number of irreducible factors, which will be large in the interesting cases. The expense of the EDF step will primarily be in creating the splitting polynomials for $A(x)$ and its factors, and the extent to which this can be done in parallel will determine the speedup achieved.

Nested parallelism with recursive calls

The structure of the EDF algorithm, where at each stage we find two non-trivial factors and apply the algorithm on them recursively to produce a tree of factors, would seem to lend itself naturally to parallel execution. OpenMP supports nested `parallel` regions which will allow us the evaluate both recursive calls simultaneously, at each step.

Doing this naively would lead to c overlapping threads and could exhaust the number of available cores. Listing 8 shows how to avoid this possibility by limiting the depth to which threads are created.

Depth	No MT mult.		MT mult.	
	Time	Speedup	Time	Speedup
2	4.10	1.33	3.39	1.61
3	3.75	1.45	3.03	1.80
4	3.70	1.47	2.93	1.86
No limit	5.56	0.98	5.97	0.91

Table 12: Comparison of the time (seconds) for the EDF step with nested parallelism, with and without the use of multi-threaded multiplication. Here $p = 101$ and $A(x) = \Phi_2(x)$, the product of all irreducibles of degree 2. For the multiplication, the first level used 12 threads and deeper levels used 2 threads.

Despite fitting neatly into the algorithm, Table 12 shows that the gains are extremely limited. This is explained by the fact that the time for generating a splitting polynomial is quadratic in the degree. If it takes time t_0 for the first split to produce two factors of roughly equal degree, then the time to split each one of these will be around $t_0/4$. The total time in serial becomes

$$t_0 + \frac{t_0}{4} + \frac{t_0}{4} + \frac{t_0}{16} + \frac{t_0}{16} + \frac{t_0}{16} + \frac{t_0}{16} + \cdots \approx t_0 \sum_{i=0}^{\infty} 2^{-i} = 2t_0.$$

But in the ideal case, when each of the splits at a given depth are done simultaneously, the time is

$$t_0 + \frac{t_0}{4} + \frac{t_0}{16} \cdots \approx t_0 \sum_{i=0}^{\infty} 4^{-i} = \frac{4}{3} t_0.$$

We can't expect speedup far exceeding $2 \cdot 3/4 = 1.5$ with this approach. This prediction agrees closely to the speedup observed when multi-threaded multiplication is disabled.

Regarding the bottom row of the table, we can see the significant overhead in creating and scheduling many more threads than the system can accommodate.

Use of the Petr-Berlekamp matrix

Calculating as in Remark 2.14, a splitting polynomial for $A(x)$ will require

$$(k-1)\lfloor \log_2 q \rfloor + k(1 + wt(q)) + \lfloor \log_2 (q-1) \rfloor - 2$$

multiplications and remainders, of which

$$(k-1)\lfloor \log_2 q \rfloor + (k-1)wt(q)$$

are needed in taking powers of q. Thus the majority of the work can be spared if the Petr-Berlekamp matrix is available. However, comparing with Equation 6, it isn't clear that computing the Petr-Berlekamp matrix will always be worthwhile. Picking out the leading terms and rearranging, we should prefer to do the exponentiation directly unless

$$d < 2N(k-2) \log_2 q.$$

This is a pessimistic estimate and doesn't consider, for example, the case when $q \ll d$ and the pre-computation takes an insignificant amount of time. However it does suggest that the matrix will be of little value when k and N are small, and an inexpensive check to this effect can be placed in the EDF routine.

Another consideration is that as part of the DDF routine, we will already have an Petr-Berlekamp matrix for a multiple of $A(x)$. We can find the matrix for $A(x)$ by taking remainders of the first d columns. Writing $R(x)$ for that multiple and using the operation counts in Remark 4.3, this will be cheaper than computing the matrix from scratch so long as $\deg R(x) < 3d$. Again, a check for this can be performed at the beginning of the routine.

The same reduction trick can be used after each split to find the matrices for the factors produced. Adding in the operations within the recursive calls which might be spared by having the matrix for $A(x)$, the estimate for when the pre-computation is worthwhile becomes

$$d < 4N(k-1) \log_2 q.$$

Table 13 compares different options for computing the matrix, in a situation where the matrix is relatively cheap and another where it's relatively expensive. If p is very small, then it would seem best to get the matrix from scratch using the shifting trick from Remark 4.4. Not only does it cut the serial runtime, but the proportion of work done in parallel is increased, as reflected by the stronger scaling. But the rule above appears to be accurate when p is larger relative to the degree. We would need N to be unrealistically big for the better speedup during the splitting to offset the cost of producing the matrix.

Threads	None	New	Unred.	Red. first	Red. all
1	5.50	4.39	6.30	4.89	4.77
2	3.44	2.31	3.14	2.38	4.12
2	3.17	1.99	2.82	2.08	3.97
6	3.13	1.90	2.80	2.07	3.89
12 (SMT)	2.91	1.82	2.67	1.87	4.67

(a) $p = 101$ and $A(x) = \Phi_2(x)$, so that $d = 10100$, $k = 2$.

Threads	None	New	Unred.	Red. first	Red. all
1	2.00	53.48	44.07	32.06	27.27
2	1.45	23.37	14.74	14.54	14.22
2	1.34	14.74	10.97	8.86	9.27
6	1.26	13.00	10.79	7.27	8.14
12 (SMT)	1.19	10.49	9.28	6.39	7.01

(b) $p = 7919$, $d = 3000$, $k = 3$.

Table 13: Comparison of the time (seconds) for the EDF step with various strategies for computing the Petr-Berlekamp matrix. From left to right, the strategies are: not using the matrix; computing the matrix at the start of the routine; using a matrix already computed for a multiple of $A(x)$ with twice the degree; producing the matrix for $A(x)$ by reducing that larger matrix; producing a new matrix for each factor using the matrix in the previous stage.

Many splitting polynomials

Seeing a benefit from doing the recursive calls in parallel requires considerable tuning in the implementation: we have to keep the number of active threads high to fully utilise the available cores, but not so high as to cause inordinate overhead in scheduling them. At its root, the problem arises because there isn't enough parallelism exposed in the early stages of the algorithm and so work must be divided out piece-meal as we progress.

A less delicate alternative would be to initially have each thread compute its own splitting polynomial, and use these to split $A(x)$ into up to 2^N factors in the first stage. We would then have enough independent tasks that they could be divvied out all at once. We essentially consolidate the first N stages of the serial algorithm into one.

Algorithm 4.8 (Many splitting polynomials)**.** Let N be the number of available threads and suppose $p \geq 3$. Suppose that we have a routine $\textsc{rand}(n)$ producing uniformly distributed polynomials of degree less than n, and a routine $\textsc{edf}(A(x), k)$ to perform equal degree factorisation in serial.

Given a squarefree polynomial $A(x) \in \mathbb{F}_q[x]$ whose irreducible factors have degree k, output the irreducible factors of $A(x)$.

```
if deg $A(x) = k$ then                    // In this case it must be irreducible by assumption
    return $A(x)$

// Produce a splitting polynomial in each thread
parallel for $i = 0, \ldots N - 1$ do
    $T_i(x) \leftarrow \textsc{rand}(\deg A(x))$
    $S_i(x) \leftarrow T_i(x)^{\frac{q^k-1}{2}} - 1 \pmod{A(x)}$

    $B_i(x) \leftarrow \gcd(S(x), A(x))$                    // $B_i(x) \mid A(x)$ and $\deg B_i(x) \approx {}^{\deg A(x)}/_2$

// Break up the list of factors (in serial) using the splitting polynomials
$L \leftarrow (A(x))$                                      // The elements of $L$ gradually get split apart
for $i = 0, \dots N-1$ do                                  // Loop through the splitting polynomials
    $M \leftarrow ()$                                      // Any empty list to hold the factors produced this iteration

    // Attempt to split each of the factors we currently have
    for $C(x)$ in $L$ do
        $D(x) \leftarrow \gcd(C(x), B_i(x))$
        $M \leftarrow (M, D(x), C(x)/D(x))$                // New factors are appended to $M$

    $L \leftarrow M$                                       // $L$ is replaced by the refinement $M$

// Have the threads do any remaining splitting separately
$N \leftarrow ()$
parallel for $C(x)$ in $L$ do
    $N \leftarrow (N, \textsc{edf}(C(x)))$                 // The appending should be done atomically

return $N$
```

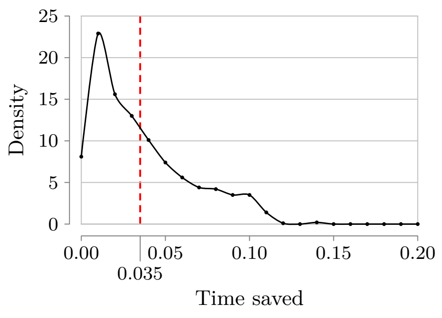
Remark 4.9. We can discard any constant factors found in the splitting step. We may also want to store the irreducible factors in a separate list so that we don't needlessly attempt to split them further. We can stop the algorithm early if we find that the list of irreducibles already has all c elements.

If t_0 is the time taken to produce the first split in the serial, then we've seen that the time for the full EDF step is approximately $2t_0$. With the above, we produce N splitting polynomials in time t_0, and if N is large enough, these will be sufficient to split apart all of the irreducible factors. Thus, in the ideal case the speedup is 2. Similarly, the performance gained by using an additional thread will be at most half of that from the previous thread. While underwhelming, this is an improvement over the nested parallelism.

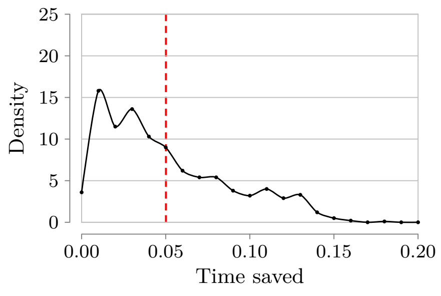
Threads	$p = 101,\ d = 10100,\ k = 2$			$p = 7919,\ d = 3000,\ k = 3$		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	4.15	1.00	1.00	1.91	1.00	1.00
2	2.37	1.75	0.88	1.34	1.42	0.71
4	1.76	2.35	0.59	1.13	1.68	0.42
6	1.84	2.25	0.38	1.46	1.31	0.22
12 (SMT)	3.82	1.08	0.09	2.12	0.90	0.07

Table 14: Comparison of the time (seconds) for the EDF step using many splitting polynomials. The Petr-Berlekamp matrix was computed and used for the polynomial on the left, but not on the right, following the measurements in Table 13.

Timings for this approach are given in Table 14. For both polynomials tested, the times



(a) $p = 5$, $d = 500$



(b) $p = 7919$, $d = 500$

Figure 4: Empirical distribution of the time (seconds) saved when the factor count is known in advance, with $n = 1000$ samples. The dotted line indicates the average time saved.

on 4 threads surpassed their respective best times when using nested parallelism. For larger p, where the matrix pre-computation isn't included in the runtime, we get close to the ideal speedup of 2. Performance noticeably regresses after a point with each, caused by the diminishing returns inherent in using multiple splitting polynomials, and the additional splitting which must subsequently be done in serial. It's apparent that symmetric multi-threading doesn't allow us to efficiently create more than one splitting polynomial at a time on a single core.

4.4 Situational improvements

Factor counting and early stopping

The factor counting made possible by Theorem 2.20 can be done independently of the factorisation, and we will have already computed Q at the beginning of the DDF step. We are free to task a thread with computing the dimension of its kernel by Gaussian elimination, and Gaussian elimination is itself easily parallelised.

On the other hand, we can keep a running count of the irreducible factors removed from $R(x)$ in Algorithm 4.5, since each $A_k(x)$ will contain exactly ${}^{\deg A_k(x)}/_k$ irreducibles. Thus, we can track the number of irreducible factors remaining and stop early once it reaches 1, at which point we know that $R(x)$ is the last factor.

Assuming that the Gaussian elimination is finished in time, the DDF step can be stopped after finding the second-largest distinct degree factor. On average this will happen at $k = 0.21d$ [10], significantly sooner than our previous stopping condition of $k > \lfloor {}^d/_2 \rfloor$.

Testing suggests that the mean time saved would be approximately 44% of runtime, independent of p and d. We should bear in mind though that this improvement won't be uniform. Figure 4 shows the large variation in the time saved, with polynomials having two large factors seeing almost no benefit, and a long tail corresponding to polynomials with one dominant factor.

Unfortunately, the factor counting is about as expensive as the main DDF step (after we compute the Petr-Berlekamp matrix), and scales similarly as we add threads. Table 15 gives timings for the factor counting and shows its effect on the overall factorisation algorithm. To provide its full value, we would like the factor count to be completed in around half the time of the DDF step. It appears that this is unlikely unless we allocate to it the majority of available threads.

Comparing with Table 11, we find that on average it's better to simply devote all of the threads to the main DDF computation. Factor counting may however be worthwhile when there is an abundance of threads and there is nothing more to be gained by using them elsewhere.

Threads	$d = 1000$			$d = 2000$		
	Time	Speedup	Prop. of DDF	Time	Speedup	Prop. of DDF
1	0.54	1.00	0.63	5.05	1.00	0.85
2	0.31	1.76	0.68	2.47	2.05	0.80
4	0.16	3.44	0.64	1.30	3.90	0.64
6	0.16	3.33	0.65	1.20	4.20	0.67
12 (SMT)	0.13	4.22	0.73	0.86	5.90	0.69

(a) Average time (seconds) to count the factors of a random polynomial, and the ratio between the time for the factor count and for the DDF step on the same number of threads, with $p = 7919$.

Threads	$d = 1000$			$d = 2000$		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency
$1 + 1$	1.47	1.59	0.79	11.88	1.52	0.76
$3 + 3$	0.63	3.69	0.62	5.18	3.49	0.58
$6 + 6$ (SMT)	0.53	4.39	0.37	4.12	4.39	0.37

(b) Average time (seconds) to factor of a random polynomial, with $p = 7919$. $N + M$ threads indicates that N threads were used in the DDF and M were used in the factor counting. All threads participated in the precomputation.

Table 15: Timings for the factor counting with various thread counts.

Dynamic concurrency

The iterations of the loops in Algorithm 2.15 can be performed totally independently, with synchronisation only needed when the irreducible factors are appended to the list for output. As in Remark 2.16, we can't expect there to be more than one costly function call in general, but nevertheless we can exploit this potential for parallelism when it arises.

Listing 9 shows how the `std::futures` from the C++ standard library can be used to dynamically spawn threads as part of the DDF routine. The same can be done for the SFF routine. A polynomial of the form $A(x) = B(x)C(x)^2$, where each of the factors is of similar degree, can see a speedup of close to 2 when the calculation is otherwise done in serial.

Deferred squarefree factorisation

The SFF step is the last component of the algorithm done entirely in serial. If early stopping is employed in Algorithm 2.6, then it will almost always end after taking a single GCD, and as such the effect on average runtime will be totally insignificant. Still, it would be pleasing to have this step parallelised, even if the gains (or lack there of) aren't large enough to be measurable.

An observation which provides some flexibility for the ordering of the SFF step is the following: if $S(x)$ is the product of the irreducible factors of $A(x)$ without multiplicity, then

$$\gcd\left(x^{q^k} - x, A(x)\right) = \prod_{i|k} S_i(x),$$

where the $S_i(x)$ are the distinct degree factors of $S(x)$. That is, we can perform the DDF factorisation step on an arbitrary polynomial and still produce the distinct degree factors, but

```
// Vector to store handles for the threads we launch
std::vector<std::future<factor_list>> edf_futures{};

for (ssize_t k = 1; k <= half_degree; ++k)
{
    /* DDF iteration omitted */

    if (factor.degree() == k) // Add to output if irreducible
        factors.emplace_back(std::move(factor), mult);

    else                      // Otherwise launch a thread
        edf_futures.push_back(std::launch::async,
                              std::async(edf_task, factor, k));
}

// Retrieve EDF results
for (auto &edf_future : edf_futures)
    factors.merge(edf_future.get());

return factors;
```

Listing 9: Dynamic thread creation in the DDF routine.

multiplicity is ignored. It's then relatively simple to recover the multiplicities, so with this we can defer the SFF step until during or after the DDF step.

Algorithm 4.10 (Deferred squarefree factorisation)**.** Given a polynomial $A(x) \in \mathbb{F}_q[x]$ of degree d and its distinct degree factorisation without multiplicity $S(x) = \prod_i^d S_i(x)$, output the distinct degree factors of the squarefree factors of $A(x)$.

```
parallel for $i = 1, \dots d$ do                    // The $S_i(x)$ can all be worked on separately

    $R_i(x) \leftarrow A(x)/S(x)$       // Product of repeated factors with their mutliplicities decremented
    $T_i(x) \leftarrow S_i(x)$                        // Product of factors of degree $i$ without multiplicity

    for $j = 1, \dots, d$ do

        $U_i(x) \leftarrow \gcd(R_i(x), T_i(x))$          // Repeated factors of degree $i$, without multiplicity
        $A_{ij}(x) \leftarrow T_i(x)/U_i(x)$                          // Picks out the non-repeated factors

        $R_i(x) \leftarrow R_i(x)/A_{ij}(x)$              // Removes non-repeated factors, decrements others
        $T_i(x) \leftarrow U_i(x)$            // Removes found factors. Equivalent to $T_i(x) \leftarrow T_i(x)/A_{ij}(x)$

return $A_{11}, \dots, A_{dd}$               // $A_{ij}$ is the product of factors of degree $i$ and multiplicity $j$
```

Remark 4.11. As before we can stop a loop once $R_i(x)$ is set to 1, and with this we perform in the average case at most one GCD for each $S_i(x)$.

Computing the GCD of polynomials with degrees n and m requires $O(nm)$ coefficient multiplications. We know that $\deg R_i(x) \leq d$, and if there are no repeated factors we also have

$$\sum_{i=0}^{d} \deg S_i(x) = d \approx \deg A'(x).$$

We should expect the total amount of work in finding these smaller GCDs to be not much greater than in computing $\gcd(A(x), A'(x))$. Also, we can skip a computation if $\deg S_i(x) > d/2$.

When integrated into the DDF step, we won't in general know $S(x)$, but we can still remove from $R_i(x)$ any factors which have been found up to that point.

Remark 4.12. A much simpler version of the algorithm in a similar vain is presented in [7, Algorithm 14.13], where the equal degree factorisation is integrated into the DDF step, and determining the multiplicities is deferred until after the irreducible factors have been found. However, this imposes the restriction that equal degree factorisation be performed as soon as a distinct degree factor is found. This would prevent us from delegating the task to a separate thread.

Table 16 compares the time spent on squarefree factorisation under the two schemes. Deferring the extraction of multiplicities is somewhat cheaper even in serial. It sees a respectable speedup on multiple threads, but when viewed as part of the total runtime, the benefit is almost unnoticeable. Deferring the SFF step like this is only sensible when q is large enough for repeated factors to be uncommon.

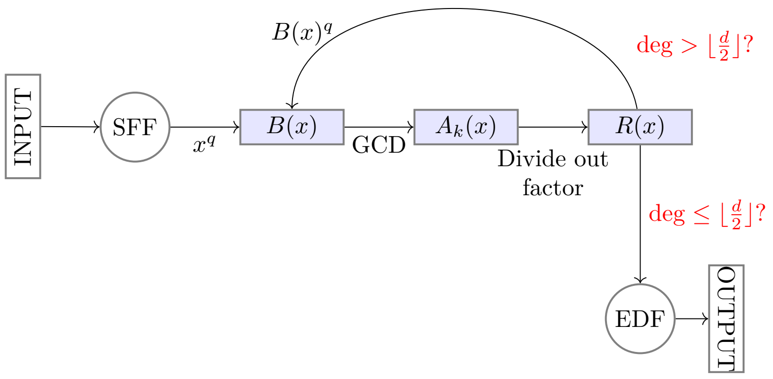
Finally, we can note that the factor counting described above remains valid due to the relaxed assumptions of Theorem 2.20.

Threads	$d = 1000$			$d = 2000$		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1 (*SSF step*)	1.32			5.50		
1	0.93	1.00	1.00	2.99	1.00	1.00
2	0.52	1.81	0.91	1.69	1.77	0.89
4	0.33	2.84	0.71	0.99	3.04	0.76
6	0.25	3.72	0.62	0.69	4.34	0.72
12 (SMT)	0.20	4.63	0.39	0.47	6.40	0.53

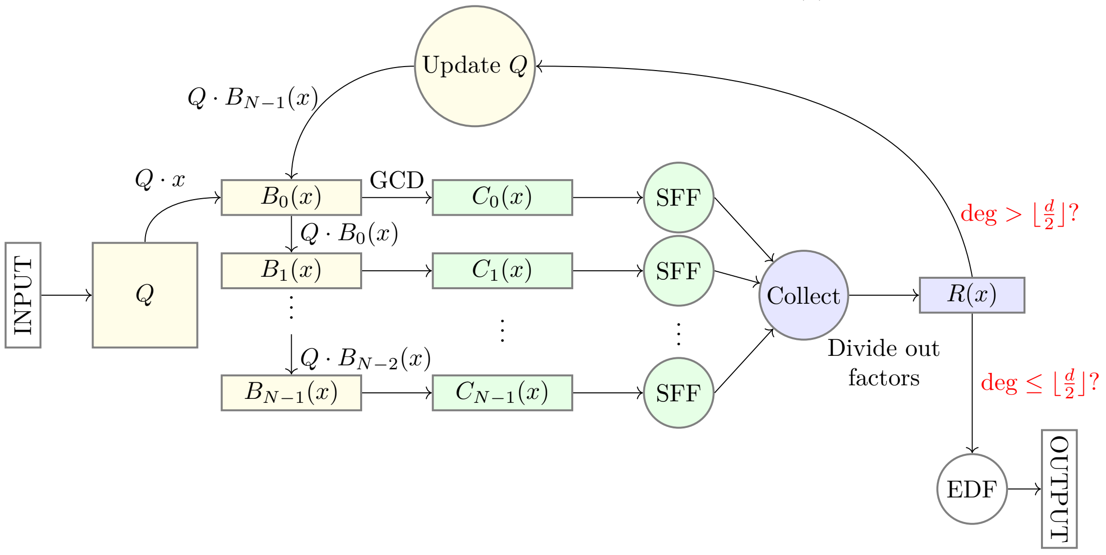
Table 16: Comparison of the average time (milliseconds) spent on the deferred squarefree factorisation of random polynomials, with $p = 7919$. The timings for the standard SFF step are denoted by (*SFF Step*). No repeated factors were encountered among the polynomials tested.

5 Parallelisation for distributed systems

Moving to the distributed context only constrains the possibilities for exploiting parallelism, so we will be concerned mostly with carrying over the ideas of Section 4, while identifying opportunities for distributing memory and for reducing inter-node communication. Some small advantage here is that there is an imposed two-tiered hierarchy of nodes and threads, which makes is easier to


$B(x)^q$
$\deg > \lfloor \frac{d}{2} \rfloor$?
INPUT
SFF
x^q
$B(x)$
GCD
$A_k(x)$
Divide out factor
$R(x)$
$\deg \leq \lfloor \frac{d}{2} \rfloor$?
EDF
OUTPUT

(a) The DDF step as described in Algorithm 2.10. At each iteration, the next power of x^p modulo $R(x)$ is computed and used to produce a distinct degree factor. This is removed from $R(x)$, and the loop repeats.



(b) The DFF step used by default in the provided implementation. The Petr-Berlekamp matrix is computed at the beginning, then used to calculate powers of x^p modulo $R(x)$, denoted as the $B_i(x)$. Each thread takes a power and uses it to produce a distinct degree factor, possibly including unwanted factors. The deferred squarefree factorisation is applied at this point. Once completed, a single thread collects the factors produced, removes duplicates, and divides them out of $R(x)$. The matrix is updated for the smaller $R(x)$ and the loop repeats.

Figure 5: Diagrams depicting the DDF stage before and after the adjustments made in Section 4. Blue indicates work done in serial, yellow work done co-operatively by the threads, and green work which is carried out independently on each thread.

speak of, and write code for, schemes involving mostly independent subdivisions of threads which work cooperatively amongst themselves.

The accompanying implementation, and the code snippets in this section, avoid interfacing with MPI directly, but instead use the bindings included with the BOOST library. The basic message passing facilities and collective calls seem to be sufficient for our goal, meaning the limited support of these bindings for later additions to the standard doesn't present any great difficulty.

In what follows, take M to be the number of nodes and N the number of threads per-node, where we assume for simplicity that each node has the same number of available threads.

5.1 Extending the shared-memory scheme

Of the adjustments described in the last section, there are three of general applicability in the average case: the pre-computation and use of the Petr-Berlekamp matrix; taking multiple GCDs at once, then removing duplicate factors; and the deferral of the SFF step until after we have the distinct degree factors. Of these, only the first two will require inter-node communication to maintain their correctness and remain reasonably performant, and only with the first are there subtleties in how this should be done.

A naive attempt may be to ignore the structure of the system we work on and instead imagine that we have a pool of MN indistinguishable threads. We can take the parallel loops which appear in the shared-memory implementation and now divide them between the available nodes, synchronising memory with all-to-all collective calls when necessary, and performing any serial sections of code on a designated root node. If this were sufficient to recreate the performance we saw previously, then there would be little need to go further.

Table 18 shows timings for such a naive distributed implementation, run on a single machine. Despite the fact that nothing is being sent over the network, the cost of synchronising and copying between processes is immediately noticeable. It appears that the absolute overhead has little dependence on p, which would be expected as in this implementation, the size of the polynomials being transferred is determined solely by their degree.

Something else worth noting is that overhead becomes smaller as a proportion of runtime as the degree of the polynomial increases. Upper bounds on the number of polynomials transferred in the DDF step are given in Table 17. Keeping in mind that the size of the transfers is proportional to d, and that the runtime is $O(d^3)$, we see that only the communication needed by the update step will remain significant when factoring polynomials of large degrees.

Routine	Root node		Other nodes	
	Send	Receive	Send	Receive
Matrix pre-computation	$M + d$	d	$1 + \frac{d}{M}$	d
Matrix multiplication	$\frac{Md}{2}$	$\frac{Md}{2}$	$\frac{d}{2}$	$\frac{d}{2}$
Duplicate clean-up	$\frac{Md}{2}$	$\frac{d}{2}$	$1 + \frac{d}{2M}$	$\frac{d}{2}$
Matrix update	$\frac{d^2}{2}$	$\frac{d^2}{2}$	$\frac{d^2}{2}$	$\frac{d^2}{2}$

Table 17: Upper bounds on the number of transfers needed in the DDF step for a polynomial of degree d, where there are M nodes and we directly carry over the shared-memory algorithm. We assume that reductions are performed solely on the root node, then broadcast to the rest. Multiplying by d and the number of bytes per coefficient gives the bounds on the amount of data transferred. The first and last rows become zero when the matrix is distributed in a striped pattern.

```
const auto reduced = b % a;        // We need degree less than deg a
Poly_mod result(a.degree() - 1); // Allocate result

// Work out offsets and how far into the polynomial we can go
const auto &[off, len] = alloc(a.degree() + 1,
                               comm.rank(), comm.size());

const auto lim = min(reduced.degree() + 1, off + len)

// First reduce on-node
#pragma omp parallel for reduction(poly_sum : result)
    for (ssize_t i = 0; i < lim; ++i)
        result += matrix[i] * reduced.coeff(i + off);

// Then reduce inter-node
mpi::all_reduce(comm, mpi::inplace_t{result}, plus<Poly_mod>{});
return result;
```

Listing 10: Matrix multiplication with a distributed matrix (in contiguous segments) and the Boost MPI bindings. Multiplication with a striped matrix is similar.

Distributing the Petr-Berlekamp matrix

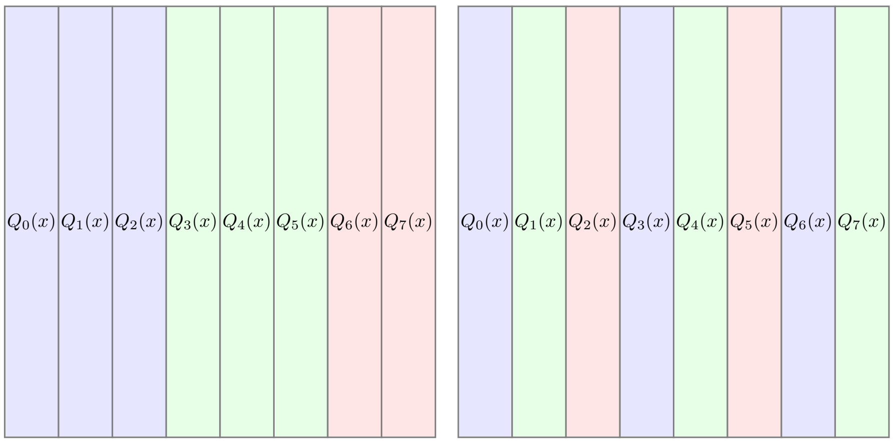
The majority of communication stems from the Petr-Berlekamp matrix and operations involving it, and it is also by far the largest component of memory usage. We would achieve both lower communication overhead and better use of distributed memory if the matrix were split across nodes, assuming that this is done without too much disruption to the rest of the algorithm.

The simplest way to realise this would be to partition the columns of the matrix evenly into contiguous segments, with each node producing its segment independently. This would eliminate any synchronisation in the pre-computation routine and would reduce the size of the matrix in memory by a factor of M. Listing 10 shows the minimal adjustments needed in the matrix multiplication routine when the matrix is distributed in this way.

Proceeding like this gives rise to balancing difficulties once the matrix is updated. The columns are discarded in order, so that the node with the highest degree columns will have its segment emptied before any other nodes have their columns removed. We might choose to periodically redistribute the matrix, in which case we would not have greatly decreased the amount of communication versus the naive implementation, or choose to tolerate having fewer nodes participate in the matrix operations as the DDF step progresses.

A solution which avoids both issues is to instead "stripe" the columns across the nodes, as in Figure 6. Now the matrix remains balanced after an update (since a column will be removed from each node before a second column is removed from any), and updates can be done without any synchronisation.

Striping the matrix does cause complications: during the pre-computation we now have to use $Q_M(x)$ for the recurrence relation, which is somewhat more expensive to calculate than $Q_1(x)$ and means that the shifting trick described in Remark 4.4 is now only relevant for $p < {}^{2d}/_{M}$; and when we apply the matrix, we no longer access the coefficients of the multiplied polynomial in sequence. The latter concern doesn't cause a measurable change in performance, but the former can be significant for small enough p. When $p < {}^{2d}/_{M}$, the total number of coefficient operations in computing a striped matrix is increased by a factor of M, effectively cancelling out



(a) Contiguous columns. (b) Striped columns.

Figure 6: Two options for distributing an Petr-Berlekamp matrix, where $M = 3$ and $d = 8$. The rightmost columns are discarded first when the matrix is updated with a smaller polynomial.

any speedup from adding nodes.

The use of a distributed, striped matrix obviates the need for inter-node communication while computing or updating the matrix, so looking again at Table 17, we find that almost all of the communication from the naive implementation has been eliminated. The benefits to runtime can be seen in Table 18. The overhead has been halved compared to the earlier attempt in most of the configurations tested, and doesn't appear to get worse as the number of processes increases.

5.2 Communication avoiding distinct degree factorisation

Communicating the factors found during the DDF step is in a sense unavoidable. They form the output of the routine, so will need to be collected to a single node eventually. Gathering them to the root at each iteration means that we only have to remove duplicates once, and it allows us to reduce $R(x)$ more frequently, hence making the later iterations cheaper, earlier. It seems unlikely that overhead saved by reordering these transfers, or by doing them asynchronously, would ever justify the extra work which would have to be done, or the increased program complexity.

More realistic is the possibility of having the nodes perform the matrix multiplications independently. We do MN matrix multiplications at each iteration of the DDF step, with each requiring global synchronisation. If we instead tasked the M nodes with N multiplications each, then we would have the same potential speedup and without the communication in-between.

If i is a node index, then one way to accomplish this is to begin by calculating

$$x^{p^i} \pmod{A(x)},$$

and subsequently have the node apply Q^M to find powers of the form

$$x^{p^{i+kM}} \pmod{A(x)}.$$

Each of the powers needed in the DDF step will be considered by some node, and this way a node can always find its next power by a single, independent matrix application.

The expense of this scheme will be in initially finding Q^M, which we call the *step matrix*. The step matrix will be the same for each node and so it may be worthwhile to have them co-operate in its calculation, although an all-to-all collective call would be needed afterwards to ensure that every node has a complete copy. The total amount of data transferred would still be less than with the distributed matrix, and would be transferred all at once at the beginning of the routine. Alternatively, we could completely avoid communication related to the matrix by having the nodes calculate it independently.

Assuming that we already have the Petr-Berlekamp matrix Q, then directly taking powers would give the step matrix in

$$d^2(\lfloor \log_2 M \rfloor + wt(M))$$

applications of Q. For us this will be equivalent to the same number of polynomial multiplications.

Instead, we could exploit the recurrence relation for Q once again, this time starting with

$$Q_1^M(x) = x^{p^M} \pmod{A(x)}$$

and using it to produce the remaining rows, This would use

$$d - 1 + (\lfloor \log_2 p \rfloor + wt(p))(\lfloor \log_2 M \rfloor + wt(M))$$

multiplications and remainders. For practical purposes the d term will dominate, and so the only the recurrence relation is actually implemented. This task can be parallelised in exactly the same way as was the pre-computation of Q, and in fact we can skip the calculation of Q itself altogether.

Table 18 presents the scheme unfavourably, albeit in an unrealistic setting where communication costs are minimal. Nevertheless, we can see the sacrifices which are made to avoid frequent synchronisations. The scaling has been greatly impacted, and the scheme does worse than the initial naive attempt with the largest process count tested.

Having the matrix applications be independent does save time as expected. For example, for $p = 7919$ and $d = 2000$, the time spent on matrix multiplication during the benchmark averaged 1.57s when using a distributed matrix, and only 0.90s under this scheme. Time is lost on the less efficient pre-computation step and the need to send a copy of the step matrix to every process afterwards.

This approach has other drawbacks. We must abandon the use of distributed memory as each node must store the full matrix locally. The range of p for which the step matrix is cheap to calculate is drastically shrunk compared to the distributed matrix, as we now require $p^M < 2d$ for the shifting trick to be useful. Finally, work is duplicated by updating the matrix separately on each node.

5.3 Distributed equal degree factorisation

Of the three means of exploiting parallelism in the EDF step given in Section 4.3, the most promising was the idea of using multiple splitting polynomials in the first stage. It showed better efficiency at low thread counts than the alternative of nested threads, while relying only on two parallel loops and thus being simple to port over. But the approach suffered from quickly diminishing returns, and couldn't take advantage of the small speedup we get from using multi-threading while generating individual splitting polynomials.

Working now in the MPI model, we can easily go from generating one polynomial per-thread in Algorithm 4.8, to one per-node. This leads to a tiered calculation for the initial stage, where the nodes carry out the largest part of the work independently and without any inter-node synchronisation, while threads within a node co-operate tightly in the polynomial multiplications. Assuming that the polynomial being factored is large enough, we will be left with around 2^M

$M \times N$	Naive				Distributed				Step matrix			
	$d = 1000$		$d = 2000$		$d = 1000$		$d = 2000$		$d = 1000$		$d = 2000$	
	Time	OH	Time	OH	Time	OH	Time	OH	Time	OH	Time	OH
1×6	0.19	0.00	1.28	0.00								
2×3	0.47	1.51	2.52	0.96	0.29	0.52	1.56	0.22	0.30	0.60	1.82	0.42
3×2	0.48	1.55	2.57	1.01	0.28	0.49	1.63	0.27	0.34	0.83	2.18	0.70
6×1	0.47	1.49	2.55	0.99	0.29	0.52	1.43	0.11	0.56	1.99	4.23	2.30

(a) $p = 5$.

$M \times N$	Naive				Distributed				Step matrix			
	$d = 1000$		$d = 2000$		$d = 1000$		$d = 2000$		$d = 1000$		$d = 2000$	
	Time	OH	Time	OH	Time	OH	Time	OH	Time	OH	Time	OH
1×6	0.60	0.00	4.57	0.00								
2×3	0.90	0.51	5.88	0.28	0.74	0.23	5.29	0.16	0.75	0.26	5.23	0.14
3×2	0.95	0.60	6.18	0.35	0.82	0.38	5.04	0.10	0.89	0.49	5.41	0.18
6×1	0.95	0.59	6.40	0.40	0.92	0.55	5.11	0.12	0.10	0.85	7.70	0.68

(b) $p = 7919$.

Table 18: Comparison of the average time (seconds) to factor a random polynomial when running MPI on a single system, with different schemes for distributing the problem between processes. Also shown is the proportion of overhead versus doing the task in a single process.

factors after all of the splitting polynomials have been used, and these can be scattered across the system for their factorisations to be completed separately.

The potential gains remain modest however. As in Remark 4.9, if t_0 is the time to produce a splitting polynomial on one node, then the time for the full factorisation done on a single node will be close to $2t_0$. By producing many splitting polynomials simultaneously, the total time can at best be brought down to t_0.

One pleasing aspect of this approach is that, unlike polynomials exchanged elsewhere in the algorithm, the splitting polynomials produced by the nodes are interchangeable and can be taken in any order. This gives us an occasion to use non-blocking MPI calls: we can take the splitting polynomials as they arrive on the root node, and use them to split apart factors while waiting for the others to appear.

We can go further to hide latency by devoting the root node solely to splitting, so that splitting polynomials are being put to use as soon as the first is ready. Ideally, the splitting will be all but finished by the time the last one arrives. Listing 11 shows how this is implemented.

The reorganisation of the algorithm is already beneficial when run on a single machine, and we see faster times for the polynomials tested in Table 14. The first polynomial can now be factored in 1.67s when $M = 3$, $N = 2$. The second has the much improved time of 0.97s when $M = 6$, $N = 1$.

5.4 Performance analysis

From the bounds in Table 17, and considering that in both of the suggested schemes we update the matrix locally, we should expect that the time saved in distributing the computation will eventually outweigh the time spent waiting on transfers and synchronisations. The point at

```
// Initialise the object which gathers the splitters from the nodes
Nonblocking_gatherer<Poly_mod> splitters(comm, root);

// If not the root, generate a splitter and send
if (comm.rank() != root)
    splitters.send(splitting_polynomial(a, k));

// On the root, work on each splitter as it arrives
else for (int i = 0; i < comm.size() - 1; ++i)
{
    const auto splitter = splitters.next();

    /* Use splitter on composite factors */
}
```

Listing 11: The latency hiding done as part of the distributed EDF step. The `Nonblocking_gatherer<>` templated class was written to abstract this communication pattern and provide a simple interface.

which this happens will depend on the cost of communication in a particular system. We will look at two distributed systems with different network architectures, comparing the effectiveness of the distributed, striped matrix and communication avoiding, step matrix approaches.

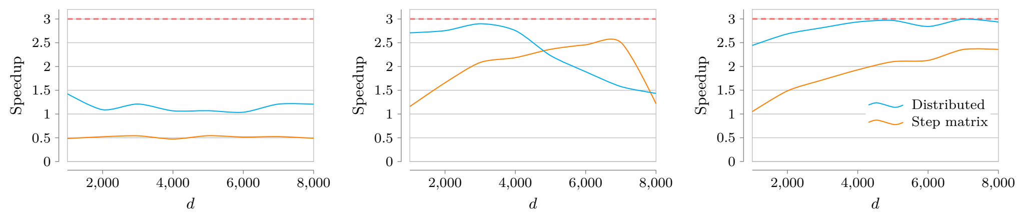
Performance on Ethernet-linked desktop PCs

The first system consists of three desktop computers using Intel Core *i5-8400* processors at 2.8 GHz, with 6 physical cores and no SMT support. The desktops are connected via Ethernet rather than a special-purpose interconnect, and so this the results here will give an idea of the performance when communication is relatively expensive.

Table 19 shows how the runtime breaks down for the two approaches and for different values of p. Although there is no great difference between them in total runtime, it is interesting to observe how the distribution of how that time is spent varies. Most strikingly, the time spent applying the matrix is halved when the calculation can be done locally. This saving is made up for by the increased time spent pre-computing and updating the matrix. In the case of small p particularly, the pre-computation of the striped matrix can make better use of the shifting trick than can the step matrix. In no instance does the removal of duplicate factors make up a meaningful portion of runtime.

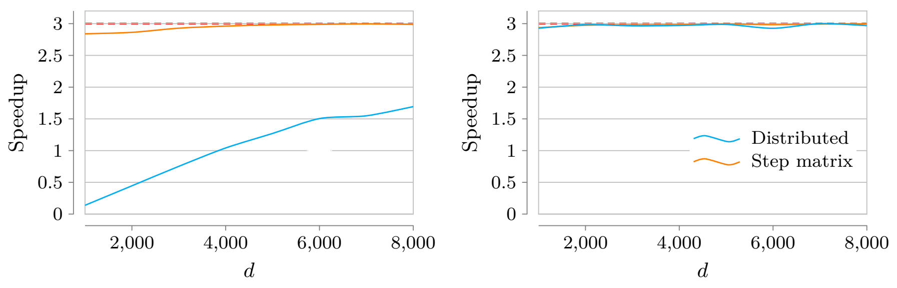
p	Scheme	Total	Pre-comp.	Matrix Mult.	GCD	Update
5	Distributed	27.28	0.11 (**0.01**)	19.45 (**0.71**)	5.50 (**0.20**)	1.68 (**0.06**)
	Step matrix	25.83	4.80 (**0.19**)	10.03 (**0.39**)	5.33 (**0.21**)	5.09 (**0.20**)
7919	Distributed	61.35	29.83 (**0.49**)	22.03 (**0.36**)	6.88 (**0.11**)	2.25 (**0.04**)
	Step matrix	63.07	36.07 (**0.57**)	12.72 (**0.20**)	6.86 (**0.11**)	6.82 (**0.11**)
$2^{31}-1$	Distributed	62.15	31.19 (**0.50**)	21.54 (**0.35**)	6.59 (**0.11**)	2.39 (**0.04**)
	Step matrix	65.79	39.71 (**0.60**)	11.84 (**0.18**)	6.54 (**0.10**)	7.05 (**0.11**)

Table 19: Breakdown of the average runtime (seconds) for the distributed factorisation, where $d = 8000$ and $M = 3$. The parenthesised figures show the proportion of total runtime.



(a) $p = 5$. (b) $p = 7919$. (c) $p = 2^{31} - 1$.

Figure 7: The speedup for the pre-computation on $M = 3$ nodes as the degree increases, for different choices of p. How well the computation of the striped matrix scales depends on the relative sizes of p, d, and M.



(a) Scaling of the matrix multiplications. (b) Scaling of the GCD calculations.

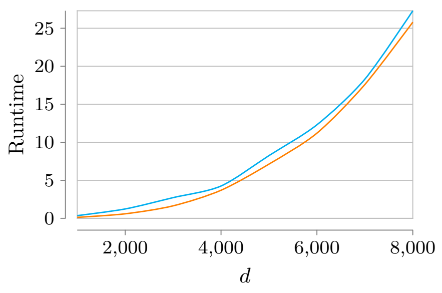
Figure 8: The speedup for the matrix multiplications and GCDs on $M = 3$ nodes as the degree increases, with $p = 2^{31} - 1$. The behaviour is independent of p.

Figure 7 demonstrates the interaction between p, d, and M in the pre-computation step for the striped matrix. When $p \ll d$, as in the left-most chart, the shifting trick applies for all sensible values of M, meaning that the work is proportional to pM and that there is practically no speedup. The middle figure shows the speedup dropping off as the value of d approaches p, so that the usefulness of shifting comes to depend of M. The last chart shows the case of $p \gg d$, where the pre-computation is done by multiplication in all cases. Here the amount of work done is independent of M and we get the desired scaling.

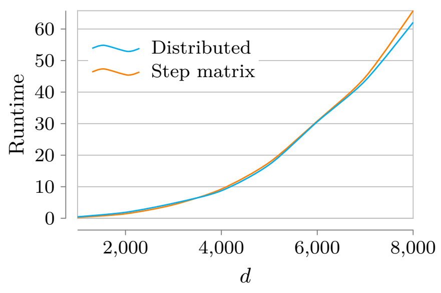
We can compare this to the behaviour of the pre-computation for the step matrix, which is closer to what we would imagine naively. A larger p leads to more work, so that the cost of copying the matrix between the nodes becomes proportionally smaller and thus the scaling better.

The speedups for the matrix multiplication and GCDs were found to be independent of p and are shown in Figure 8. We get close to perfect speedup when using the step matrix for exponentiation, since there's no overhead from communication, but the scaling with the distributed matrix is much weaker. We can notice that the gap closes as d increases, suggesting that eventually there may be no discernible difference. Under both schemes, the GCDs are done by each thread independently and suffer no overhead, reflected in the chart by the constant, ideal speedup.

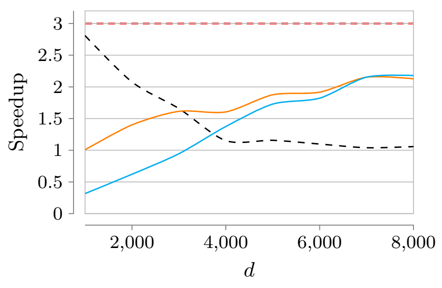
Finally, the overall runtimes are compared in Figure 9. As could be predicted, the advantages of using the step matrix become less compelling as the problem size increases. With communication being less of a performance consideration when factoring large polynomials, the reduced memory usage of the distributed matrix is likely to make it the more appealing approach.



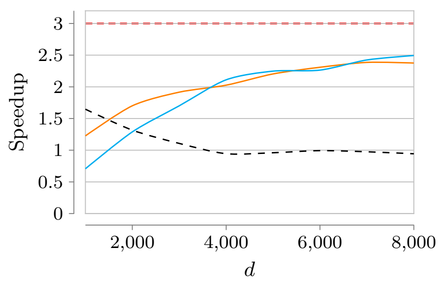
(a) The total runtime with $p = 5$.



(b) The total runtime with $p = 2^{31} - 1$.



(c) The speedup with $p = 5$.



(d) The speedup with $p = 2^{31} - 1$.

Figure 9: The total runtime (seconds) on $M = 3$ nodes as the degree increases. The black dashed line gives the ratio of the runtime with the distributed approach against that with the step matrix approach. The step matrix is superior for small polynomials where communication is more significant, but the advantage gradually diminishes

Performance on a dedicated cluster

The second system we will look at consists of 20 nodes, each with an Intel Xeon *E5-1620 v3* processor at 3.5 GHz, having 4 cores and supporting 2-way SMT. Having seen how the components of the computation are affected by the need for inter-node communication, our primary interest here will be to see how performance scales to a larger number of nodes.

Figure 10 shows the speedup on this system for each of the two schemes for a fixed problem size. Clearly, each has some significant aspect of the computation which scales poorly. The best speedup observed was 5.52 on 14 nodes for the distributed matrix scheme, when p is large enough for the pre-computation to be both time consuming and effectively parallelised.

The weak point for the distributed matrix is the matrix multiplication used for exponentiation, which sees a peak speedup of 1.74 on only $M = 3$ nodes, then trailing off as the inherent expense of performing collective operations on a large number of nodes counteracts the decrease in the amount of work per-node. The step matrix approach, in contrast, sees perfect scaling for the exponentiation, but struggles elsewhere as more work is duplicated to avoid communication.

A stark difference not visible from the charts is that of memory usage. To factor a polynomial of degree $d = 16000$ over $M = 16$ nodes, the distributed matrix approach uses 164MB of memory in total and completes the calculation in 216 seconds, whereas the step matrix approach uses 2.01GB and takes 296 seconds.

Looking now at distributed equal degree factorisation, Figure 11 confirms the prediction that speedup against a single node will be close to 2[8]. The speedup is slightly better when splitting a large number of linear factors, but already at $k = 5$ it regresses to the expected limit.

Performing the splitting on the root node as the splitting polynomials arrive is effective in hiding the time spent waiting for nodes' results after the first, but neither of these form a significant part of runtime. For $k = 5$, they constituted less than 5% of the total time.

6 Conclusions

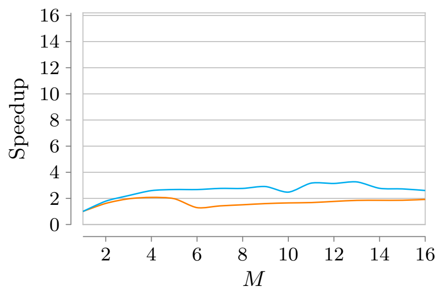
To summarise, we have found that by rearranging and otherwise adjusting the "textbook" Cantor-Zassenhaus algorithm, we can expose parallelism at a higher level than at the underling polynomial arithmetic, and thereby see significant performance improvements when run on parallel systems. The algorithmic complexity remains unchanged, but work has been separated to the extent that we can observe close to ideal speedups, at least when memory is shared and when the degree is large compared to the number of threads.

The basic operations – addition, scalar multiplication and polynomial multiplication – appear from their definitions to be trivially parallelisable, but in fact the first two are too insubstantial to see a benefit from multi-threading. The largest single adjustment is to avoid the majority of these operations by replacing the modular exponentiation with multiplication by the Petr-Berlekamp matrix. A simple trick means that the pre-computation of the matrix can be done with (asymptotically) optimal speedup over the serial, recurrence relation method.

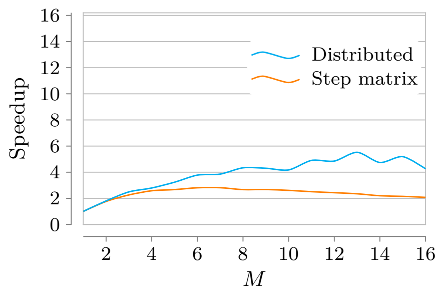
The other major change is in extracting multiple distinct degree factors simultaneously. This is sub-optimal if done in serial, as some additional work must be done to remove duplicate factors and opportunities to reduce the polynomial are missed. On average though, we see impressive gains and the approach allows performance to scale beyond a small number of threads.

A curiosity, which is of limited help in practice, is the fact that squarefree factorisation can be done in parallel if delayed until after the distinct degree factorisation. Similarly, the idea of spawning a thread to complete the factorisation of each non-trivial factor as they are found is

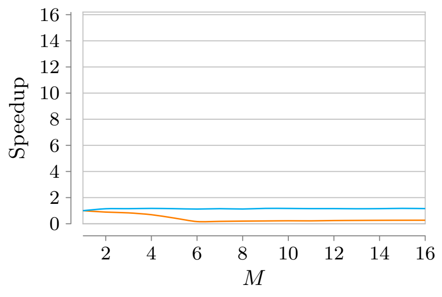
[8]The speedup against a single thread will be slightly greater. Table 12 shows there is some benefit to multi-threaded multiplication, which we avail of with this scheme.



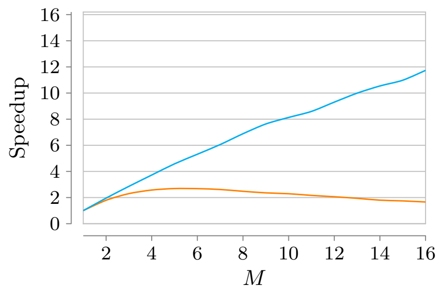
(a) Full calculation with $p = 5$.



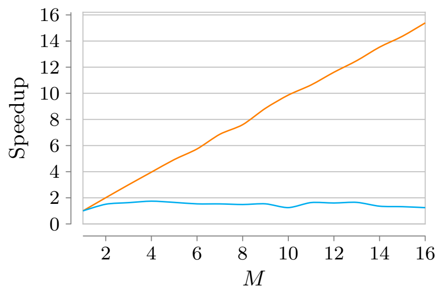
(b) Full calculation with $p = 2^{31} - 1$.



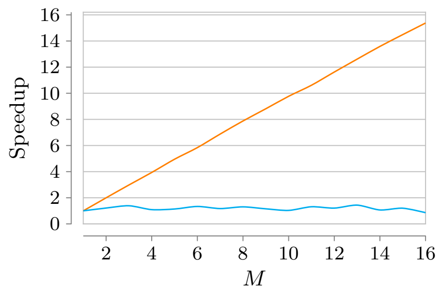
(c) Pre-computation with $p = 5$.



(d) Pre-computation with $p = 2^{31} - 1$.

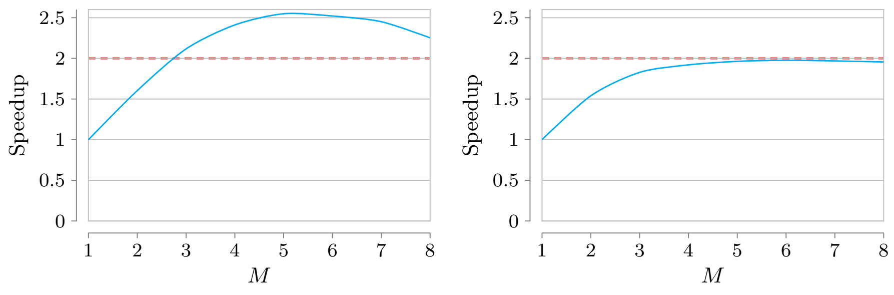


(e) Exponentiation with $p = 5$.



(f) Exponentiation with $p = 2^{31} - 1$.

Figure 10: The speedup for different parts of the computation for $d = 8000$ as M increases, with small and large choices of p.



(a) $k = 1$, $d = 2000$. (b) $k = 5$, $d = 10000$.

Figure 11: The speedup for distributed equal degree factorisation as M increases, for different degrees of the irreducible factors. Here $p = 7919$, although the behaviour is independent of p. The number of nodes given is the number of nodes producing splitting polynomials. In each case there is an additional root node set aside to collect them and split factors.

attractive, but the vast majority of factors peeled off in the average case are already irreducible, and the potential benefits for smooth polynomials are underwhelming.

Finding the total count of factors and using this information to stop early is another idea which may be useful in general. The potential time savings are enticing, and the factor count may be done without any synchronisation with the main factorisation routine. On average, however, it seems better to simply devote more threads to factorisation.

When adapting the factorisation for distributed systems, we encounter a trade-off between memory efficiency and the cost of finding and updating the Petr-Berlekamp matrix on one hand, and reducing communication overhead on the other. Neither of the schemes considered scaled well beyond a small number of nodes with the problem size tested, as for each there are aspects of the calculation whose cost remains near constant as the number of nodes increases.

Runtime is cubic in the degree of the polynomial on average, while the number of synchronisations scales linearly and the amount of data sent scales quadratically, so that in both senses the communication becomes less significant for larger polynomials. The limiting factor for the distributed matrix scheme was the need to synchronise each time the matrix was applied, so we should expect sufficiently large polynomials to achieve near ideal speedup on a given number of nodes.

Finally, we saw that the above changes have little relevance in factoring smooth polynomials. A tiered approach, in which threads co-operate closely but where nodes work largely independently, managed to halve the runtime. Use of the Petr-Berlekamp matrix can also reduce the runtime somewhat, if it is already available or if p is small enough for the computation to be insignificant.

Further work

The most obvious improvement to the provided implementation would be to employ asymptotically fast algorithms for multiplication, remainders, and GCDs when dealing with large polynomials. In principle, the techniques discussed here would remain relevant, with the possible exception of using the Petr-Berlekamp matrix in place of modular exponentiation. With fast multiplication, the complexity of powering a polynomial directly becomes lower than that of applying a matrix. However, the crossover point will depend on p, and is likely to be beyond the point at which factorisation is practical.

We saw that for small p, the pre-computation of a striped, distributed matrix can't be effectively parallelised. Another improvement may be to switch to distributing the matrix in contiguous segments for this case. However, it's not clear that this would always be useful, due to the complications arising when the matrix is updated.

The observed relative slowness of the Gaussian elimination used for factor counting may be due to the quality of the implementation. A tuned Gaussian elimination routine from a specialised finite field linear algebra library might make factor counting a more appealing use of resources. In the distributed context, it may be worth investigating whether adapting existing techniques for distributed LU factorisation can yield better scaling than what we've seen for the DDF step. We could use this as a source of further gains, once the potential for speedup in the factorisation itself has been exhausted.

References

[1] Olaf Bonorden et al. "Factoring a binary polynomial of degree over one million". In: *ACM SIGSAM Bulletin* 35 (Apr. 2001). DOI: 10.1145/504331.504333.

[2] Allan Borodin, Joachim von zur Gathen, and John Hopcroft. "Fast parallel matrix and GCD computations". In: *Information and Control* 52.3 (1982), pp. 241–256. ISSN: 0019-9958.

[3] David G. Cantor and Hans Zassenhaus. "A New Algorithm for Factoring Polynomials Over Finite Fields". In: *Mathematics of Computation* 36.154 (1981), pp. 587–592. ISSN: 00255718, 10886842. URL: http://www.jstor.org/stable/2007663.

[4] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer Publishing Company, Incorporated, 2010. ISBN: 3642081428.

[5] Henri Cohen et al. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. 2nd ed. Chapman & Hall/CRC, 2012. ISBN: 1439840008.

[6] Agner Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*. 2022. URL: https://www.agner.org/optimize/instruction_tables.pdf.

[7] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. 3rd ed. Cambridge University Press, 2013. DOI: 10.1017/CBO9781139856065.

[8] Joachim von zur Gathen. "Parallel Algorithms for Algebraic Problems". In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. STOC '83. New York, NY, USA: Association for Computing Machinery, 1983, pp. 17–23. ISBN: 0897910990. DOI: 10.1145/800061.808728.

[9] Joris Hoeven, Grégoire Lecerf, and Guillaume Quintin. "Modular SIMD arithmetic in Mathemagix". In: *ACM Transactions on Mathematical Software* 43 (July 2014). DOI: 10.1145/2876503.

[10] Daniel Panario. *Random Polynomials over Finite Fields: Statistics and Algorithms*. URL: https://www.stat.purdue.edu/~mdw/ChapterIntroductions/PolynomialsDanielPanario.pdf.

[11] Victor Shoup. *Thread boosting and polynomial factorization in NTL*. 2016. URL: https://libntl.org/boost.pdf.

[12] The PARI Group. *PARI/GP*. Univ. Bordeaux, 2022.

[13] Victor Shoup. *NTL: A Library for Doing Number Theory*. 2021.