A parallel implementation for polynomial multiplication modulo a prime.

Marshall Law
Department of Mathematics
Simon Fraser University
Burnaby, B.C. Canada.
mylaw@sfu.ca.

Michael Monagan
Department of Mathematics
Simon Fraser University
Burnaby, B.C. Canada.
mmonagan@cecm.sfu.ca.

ABSTRACT

We present a parallel implementation in Cilk C of a modular algorithm for multiplying two polynomials in $\mathbb{Z}_q[x]$ for integer $q > 1$, for multi-core computers. Our algorithm uses Chinese remaindering. It multiplies modulo primes $p_1, p_2, ...$ in parallel and uses a parallel FFT for each prime. Our software multiplies two polynomials of degree 10^9 modulo a 32 bit integer q in 83 seconds on a 20 core computer.

Categories and Subject Descriptors

F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*Computation of transforms, Computations on polynomials*; I.1.2 [**Symbolic and Algebraic Manipulation**]: Algebraic Algorithms

Keywords

Fast Fourier Transform, Parallel Algorithms

1. INTRODUCTION

We are building a parallel library for arithmetic in $\mathbb{Z}_q[x]$. The new library will be used by Maple for polynomial factorization in $\mathbb{Z}[x]$ and $\mathbb{Z}_q[x]$ for prime q and for multivariate polynomial GCD computation over $\mathbb{Z}$ where modular algorithms are used. It will replace an older C library, the `modp1` library, that was developed by Monagan in [16] to improve Maple's polynomial factorization performance.

The parallel library will need to support a range of modulus sizes. The algorithm used for factorization in $\mathbb{Z}[x]$ uses small primes, as small as $q = 3$. The primes needed for multivariate polynomial GCD computation in [10] include 32 bit primes, 64 bit primes and also 128 bit primes. The target architecture for the new library is multi-core computers because that is what most Maple users are using.

A core routine for the library is multiplication in $\mathbb{Z}_q[x]$ because asymptotically fast algorithms for division, polynomial interpolation, the Euclidean algorithm, etc., can be reduced to fast multiplication. The question we address in this paper is how should we multiply in $\mathbb{Z}_q[x]$ on a multi-core computer for a range of prime sizes?

Let us fix notation. Let $a(x) = \sum_{i=0}^{da} a_i x^i$ and $b(x) = \sum_{i=0}^{db} b_i x^i$ be polynomials in $\mathbb{Z}_q[x]$ for an integer $q > 1$. Let $c(x) = a(x) \times b(x) = \sum_{i=0}^{da+db} c_i x^i$ be their product.

Let $u(x) = a(x) \times b(x)$ without reduction mod q, that is, treating the coefficients of $a(x)$ and $b(x)$ as integers. Let $\|u(x)\|_\infty$ denote the magnitude of the largest coefficient of $u(x)$. Then $\|u(x)\|_\infty < q^2 \min(da + 1, db + 1)$.

Kronecker substitution

We outline two approaches for computing $c(x)$ fast. Both are described in [6]. Currently, Maple and Magma both reduce multiplication in $\mathbb{Z}_q[x]$ to multiplication of two long integers. Both use the GMP integer arithmetic package [7] and hence exploit GMP's asymptotically fast integer multiplication algorithm. Let B be the base of the integer representation. In GMP $B = 2^{64}$ on a 64 bit machine. The algorithm is

```
1 Pick the smallest integer $k$ with $B^k > \|u(x)\|_\infty$.
2 Construct long integers $a(B)$ and $b(B)$.
3 Multiply them: $c(B) := a(B) \times b(B)$.
4 Write $c(B) = \sum_{i=0}^{da+db} \bar{c}_i B^{ik}$ for $0 \le \bar{c}_i < B^k$.
  Compute $c_i := \bar{c}_i \mod q$ for $i = 0$ to $da + db$.
5 Output $c(x) = \sum_{i=0}^{da+db} c_i x^i$.
```

For fixed q, steps 2, 4 and 5 take time almost linear in $da + db$. Step 3 is a large integer multiplication of size $k\,da$ by $k\,db$ words. Maple and Magma both use GMP 5.0 for integer arithmetic. GMP is using the Schönhage-Strassen algorithm [18] for long integer multiplication which has complexity $O(n \log n \log\log n)$ where $n = k\,da + k\,db$.

This method is easy to implement. It utilizes the GMP implementation of the Schönhage Strassen algorithm which has been optimized over many years and it scales well for different sizes of q, but, it is not parallelized.

Chinese remaindering

Another approach to fast multiplication in $\mathbb{Z}_q[x]$ (also in $\mathbb{Z}[x]$) is to multiply $u(x) = a(x) \times b(x)$ in $\mathbb{Z}[x]$ using Chinese remaindering then reduce the coefficients in the product u modulo q. Let $n = 2^k$ be the first power of 2 greater than $\deg a + \deg b$. Pick m primes $p_1, p_2, ..., p_m$ which satisfy

(i) $M = \Pi_{i=1}^m p_i > \|u(x)\|_\infty$,

(ii) $n | (p_i - 1)$ for $1 \le i \le m$, and

(iii) p_i is a word size prime, small enough so that arithmetic in $\mathbb{Z}_{p_i}$ can be done using the hardware of the machine.

PASCO '15, July 10 - 12, 2015, Bath, United Kingdom
© 2015 ACM. ISBN 978-1-4503-3599-7/15/07...$15.00
DOI: http://dx.doi.org/10.1145/2790282.2790293

The modular multiplication algorithm is simple.

1 Multiply $c_i(x) := a(x) \times b(x) \mod p_i$ for each prime.
2 Letting $c_i(x) = \sum_{j=0}^{da+db} c_{ij} x^j$, for each j find $u_j \in \mathbb{Z}$ using Chinese remaindering such that $0 \le u_j < M$ and $u_j \equiv c_{ij} \mod p_i$ for $1 \le i \le m$.
3 Recover the coefficients of $c(x)$ from $c_j = u_j \mod q$.

Condition (ii) means the FFT can be used in step 1 to multiply $a(x) \times b(x)$ inside $\mathbb{Z}_{p_i}$ using $O(n \log n)$ multiplications. We chose this approach to parallelize because there is an easy factor of m in parallel speedup available; multiply modulo each prime in parallel. Also, the Chinese remaindering is easily parallelized. Our C implementation uses the following 31 bit primes

$$\begin{aligned} p_1 &= 2013265921 = 15 \times 2^{27} + 1 \\ p_2 &= 1811939329 = 9 \times 2^{26} + 1 \\ p_3 &= 469762049 = 7 \times 2^{26} + 1 \\ p_4 &= 2113929217 = 63 \times 2^{25} + 1 \\ p_5 &= 1711276033 = 51 \times 2^{25} + 1. \end{aligned}$$

If q is at most 32 bits, we can multiply polynomials whose product $c(x)$ has $\deg c < 2^{26}$ using the first three primes. We have $M = p_1 p_2 p_3$ is 90.5 bits which is large enough to recover $c(x)$ as $\|u(x)\|_\infty < 2^{26} q^2 < 2^{90}$. If q is 63 bits and the $c(x)$ has degree $\deg c < 2^{25}$ then we use all five 31 bit primes. Otherwise our C implementation uses sufficiently many 63 bit primes to satisfy conditions (i) and (ii). For 63 bit primes we need to code multiplication and division by p in assembler. One advantage of using 31 bit integer primes instead of 63 bit primes is that we get more (easy) parallelism. Another reason is that a 32 bit integer multiplication in $\mathbb{Z}_p$ is more than twice as fast as 64 bits.

Outline of paper

Throughout the paper p will refer to one of the primes $p_1, p_2, \ldots, p_m$. To implement multiplication in $\mathbb{Z}_p[x]$ we need to implement the discrete FFT. And to parallelize multiplication in $\mathbb{Z}_q[x]$ we need also to parallelize the FFT since otherwise parallel speedup is limited to a factor of m.

In Section 2 we will give details of our implementation of the FFT and how we parallelized it. We are using Cilk C which allows us to program a serial algorithm in C then add Cilk directives to parallelize the code. This is easy when the parallel tasks are data independent.

When we implemented the modular algorithm we hoped that our code would come close to the speed of GMP on one core so that we could beat GMP by at least a factor of two on a quad-core desktop. The data in Section 4 shows that for a 31 bit prime q we are about 3 times faster than GMP on one core. With parallelization we are about 9 times faster on a quad-core desktop. To achieve this we implemented many optimizations, most of which are not new. We present the most useful ones in Section 3.

In Section 4 we compare our multiplication to the Kronecker substitution in Maple and Magma and measure the effect of turning each optimization off and the parallel speedup that we get on a 20 core machine. Space becomes an issue for polynomials of very large degree. Our software can compute the product of two polynomials of degree one billion modulo a 32 prime on a machine with 128 gigabytes of RAM which we think is a size record. We conclude by describing two applications where we need to multiply polynomials of degree one million or more.

Related work

The general literature on the FFT is very large. The reader may find the text [3] by Chu and George helpful. There are many floating point implementations for FFTs over $\mathbb{R}$ and $\mathbb{C}$. We found the paper [5] by Frigo and Johnson helpful. It gives implementation details of the algorithms used in the popular FFTW library (see http::/www.fftw.org).

In 1993 Shoup [19] implemented the Chinese remaindering approach using the FFT to factor polynomials in $\mathbb{Z}_q[x]$. Shoup's 1995 paper [20] includes many details that we found helpful. In [2] Chmielowiec parallelized the modular algorithm (on the primes only) for multiplication in $\mathbb{Z}[x]$ using Open MP. His experiments on quad-core machines have polynomials with 512 bit coefficients so m is large enough so that no parallelization of the FFT is needed.

Several Computer Algebra systems use an FFT based implementation to multiply polynomials in $\mathbb{Z}_q[x]$, for example, Victor Shoup's NTL package [21]. We note that Shoup has put parallelizing the modular algorithm (on the primes) on his wish list for NTL in late 2014.
See http://www.shoup.net/ntl/doc/tour-changes.html.

The Maple library `modpn` developed by Xi, Moreno Maza, Rasheed, and Schost [12] includes a serial FFT for 32 bit primes. It was developed to multiply polynomials modulo a triangular set. Moreno Maza and Xie in [14] claim to be the first to implement a parallel FFT for multi-core computers for multiplication in $\mathbb{Z}_p[x]$. They used Cilk C++. The goal of their work is to apply FFT techniques to fast normal form computation modulo a triangular set.

The Spiral project [17] has developed libraries for the discrete FFT for various architectures including multi-core computers. The paper [4] gives an overview of many variants of the FFT and optimizations and discusses floating point codes which use vector instruction sets. The codes that are generated are large. One goal of the Spiral project (see http://www.spiral.net) is to automate the creation and tuning of FFT libraries [11].

In [9] van der Hoeven developed a *truncated FFT* that saves up to a factor of 2 in the number of multiplications done by the FFT when $2^{k-1} \le \deg a + \deg b \ll 2^k$. In [8] Harvey presents a new serial FFT algorithm which combines van der Hoeven's truncated FFT with David Bailey's cache-friendly FFT [1]. Meng and Johnson [13] improve on Harvey's work by incorporating vector instructions and multi-threading into the truncated FFT. The implementation is derived and tuned using the Spiral system for code generation. This library currently uses 16 bit primes because of the limitation of integer SSE vector instructions. We have not implemented the truncated FFT.

2. A PARALLEL FFT

Let $a(x) = \sum_{i=0}^{d} a_i x^i$ be a polynomial with coefficients $a_i \in \mathbb{Z}_p$ and let ω be a primitive n'th root of unity in $\mathbb{Z}_p$ with $n = 2^k$ and $n > d$. Figure 1 is an in-place recursive Radix 2 discrete FFT that outputs

$$F = [a(1), a(\omega), a(\omega^2), \ldots, a(\omega^{n-1})]$$

the discrete Fourier transform of $a(x)$. The input $A = [a_0, a_1, ..., a_d, 0, 0, ..., 0]$ is an array of size n of the coefficients of $a(x)$ padded with zeroes, $W = [1, \omega, \omega^2, \ldots, \omega^{n/2-1}]$ is an array of size $n/2$, and T is an array of size n of working memory.

```
void fft1( int *A, int n, int stride,
           int *W, int p, int *T )
{  int i,n2,t;
   if( n==1 ) return;
   n2 = n/2;
   // Step 1 : permutation
   for( i=0; i<n2; i++ ) {
      T[   i] = A[2*i];
      T[n2+i] = A[2*i+1]; }
   // Step 2 : recursive calls
   fft1( T,    n2, stride+1, W, p, A );
   fft1( T+n2, n2, stride+1, W, p, A+n2 );
   // Step 3 : arithmetic (butterfly)
   for( i=0; i<n2; i++ ) {
      t = mulmod(W[i<<stride],T[n2+i],p);
      A[   i] = addmod(T[i],t,p);
      A[n2+i] = submod(T[i],t,p); }
   return;
}
```

Figure 1: C code for the FFT

Step 1 moves the even coefficients of $a(x)$ to the first half of T and the odd coefficients of $a(x)$ to the second half so that $T = [a_0, a_2, ..., a_{\frac{n}{2}-2}, a_1, a_3, \ldots, a_{\frac{n}{2}-1}]$. The two recursive calls on the even and odd coefficients of $a(x)$ use the first half and second half of A as temporary space. They are independent and can be executed in parallel. The two for loops may also be parallelized.

Letting $F(n)$ be the number of multiplications done in $\mathbb{Z}_p$ done by this FFT. We have $F(n) = 2F(\frac{n}{2}) + \frac{n}{2}$ multiplications for $n > 1$. Solving the recurrence with $F(1) = 0$ we obtain $F(n) = \frac{n}{2}\log_2 n$ multiplications (see Theorem 8.15 in [6]).

There are four immediate issues that we address here.

2.1 Memory access of W

The algorithm accesses the elements of W in step 3. It does so once sequentially as $1, \omega, \omega^2, \omega^3, \ldots$ with stride = 0. In the two recursive calls where stride = 1, it accesses the even powers $1, \omega^2, \omega^4, \ldots$ twice. If $n = 2^{20}$, in the middle of the recursion, when $n = 1024$ it accesses the powers $\omega^0, \omega^n, \omega^{2n}, \ldots$ 1024 times. This will cause severe cache misses. Our solution is to append to W all power sequences of ω that we need. That is we construct

$$\begin{aligned} W = [\ & 1, \omega^1, \omega^2, ..., \omega^{\frac{n}{2}-1}, \\ & 1, \omega^2, \omega^4, ..., \omega^{\frac{n}{2}-2}, \\ & 1, \omega^4, \omega^{16}, ..., \omega^{\frac{n}{2}-4}, \quad ..., \quad 1, \omega^{n/4}, 1, 0 \quad] \end{aligned}$$

with a 0 at the end (not used) to make the length a power of 2. This requires no additional multiplications in $\mathbb{Z}_p$ but doubles the space needed for W from $\frac{n}{2}$ to n. We pass a pointer in the recursive calls to $W + \frac{n}{2}$ as shown below so that the powers ω are always accessed sequentially in memory.

```
void fft1( int *A, int n, int *W, int p, int *T )
{  ...
   // Step 2 : recursive calls
   fft1( T,    n2, W+n2, p, A );
   fft1( T+n2, n2, W+n2, p, A+n2 );
   // Step 3 : arithmetic
   for( i=0; i<n2; i++ ) {
      t = mulmod(W[i],T[n2+i],p);
      A[   i] = addmod(T[i],t,p);
      A[n2+i] = submod(T[i],t,p); }
   return;
}
```

2.2 The recursion base

This recursive implementation of the FFT has a locality advantage over an iterative version but this comes at a cost of recursion overhead in both the serial version with function call overhead and also the parallel versions with Cilk overhead. Rather than using a non-recursive version of the FFT, the serial overhead can be reduced by hard coding the base for either $n = 2$, $n = 4$, $n = 8$, etc. For example for $n = 2$ we may use

```
if( n==2 ) { int t1,t2;
    t1 = addmod( A[0], A[1], p );
    t2 = submod( A[0], A[1], p );
    A[0] = t1; A[1] = t2; return;
}
```

Here we have eliminated multiplication by $\omega = 1$ which saves $\frac{n}{2}$ multiplications. If we do this for $n = 4$ we can eliminate more recursive calls, and save an additional $\frac{n}{4}$ multiplications. We found this to be effective also for $n = 8$. NTL [21] does this up to $n = 4$.

Naively parallelizing the recursive calls in `fft1` in Cilk C by doing

```
spawn fft1( T,    n2, W+n2, p, A    );
spawn fft1( T+n2, n2, W+n2, p, A+n2 );
sync; // wait till both tasks complete
```

means all recursive calls can be executed in parallel. However, the work in the FFT for n under 1000 is too small to parallelize in Cilk because the Cilk overhead becomes significant. It is better to set a cutoff below which the FFT is simply executed serially as shown below. The optimal value for the cutoff can be determined by experiment.

```
#define CUTOFF 100000
cilk void parfft1( int *A, int n,
                   int *W, int p, int *T )
{  int i,n2,t;
   // Step 0 : for small n use serial code
   if( n<CUTOFF ) return fft1( A,n,W,p,T );
   ...
   // Step 2 : recursive calls
   spawn parfft1( T,    n2, W+n2, p, A    );
   spawn parfft1( T+n2, n2, W+n2, p, A+n2 );
   sync;
   ...
   return;
}
```

2.3 Arithmetic

If p is a 31 bit prime, to multiply $a \times b \mod p$ we can in C use a 64 bit integer, a long int, to hold the 62 bit product $a \times b$ before dividing by p. Multiplication is

```
inline int mulmod(int a, int b, int p)
{ return (long) a * b % p; }
```

The cost of the hardware division instruction, however, is prohibitive. On an Intel E5-2680 v2 processor, the integer multiplication $a \times b$ costs 3 cycles but one division of ab by p costs 21 cycles. There are many papers in the literature that replace the division by multiplications. We are using the algorithm described by Möller and Torbjorn in [15]. We explain the basic idea here.

Let $c = ab$ be the 62 bit integer product. We first compute the quotient q of $c \div p$ then the remainder r using $r = c - pq$. The idea to compute the quotient is to precompute $s = 2^k/p$ to 64 bits of precision then the quotient is approximately $(s \times c)/2^{64}$. This reduces $c \mod p$ to two multiplications plus some shifts and other cheap operations. Our C implementation for computing $ab \mod p$ takes 7.6 cycles in an unrolled loop for a 31 bit prime p.

We noticed, however, that the `gcc` compiler will automatically replace division by p with multiplications if p is hard-coded and that the code generated by the compiler takes 3.8 cycles in an unrolled loop. Therefore we have compiled copies of the code for each prime with the primes hardwired in the code. We illustrate for p_1.

```
void fft1p1( int *A, int n, int *W, int *T )
{  int i,n2,t,p;
   if( n==1 ) return;
   p = 2013265921; // p1 = 15 x 2^27 + 1
   ...
}
```

If p is a 31 bit prime and we use signed 32 bit integers then we can encode addition $a + b$ and subtraction $a - b$ in $\mathbb{Z}_p$ without overflow as follows

```
inline int addmod(int a, int b, int p)
{  int t = a-p+b;
   if( t<0 ) t += p;
   return t;
}
inline int submod(int a, int b, int p)
{  int t = a-b;
   if( t<0 ) t += p;
   return t;
}
```

In step 3, the heart of the FFT, because there is a subtraction and an addition for every multiplication, these branches have a huge impact on performance. Replacing the branch `if( t<0 ) t+= p;` with `t += (t>>31) & p;` on modern processors, speeds up the entire FFT by a factor of 2. This was a surprise to us.

2.4 Space used

Figure 2 shows how to multiply $c(x) = a(x) \times b(x) \mod p$ using the FFT. The algorithm computes $c(x)$ using

$$n^{-1}FFT(\ FFT(a, \omega, n) \times FFT(b, \omega, n),\ \omega^{-1}, n\ )$$

where $\times$ denotes point-wise multiplication.
If we want to execute the two forward transforms in steps 4 and 5 in parallel only W can be shared so we will need arrays for each of A, B, W, T and C of size n thus $5n$ units of storage. If we do this for m primes in parallel the total space needed is $5mn$ units of storage. To multiply two polynomials of degree one billion, the FFT will use $n = 2^{31}$. Since there are no 32 bit primes p with $n|p-1$ we use two 63 bit primes. Thus the total space needed is $10n$ words which is 160 gigabytes. In Section 3.1 we will reduce the space to $3mn$ units of storage and in Section 3.4 further to $2\frac{2}{3}mn$.

```
1 Create $W$, the array of powers of $\omega$.
2 Create $A = [a_0, a_1, \ldots, a_{da}, 0, 0, ..., 0] \in \mathbb{Z}_p^n$.
3 Create $B = [b_0, b_1, \ldots, b_{da}, 0, 0, ..., 0] \in \mathbb{Z}_p^n$.
4 Compute fft1$(A, \omega, n, W, T, p)$
5 Compute fft1$(B, \omega, n, W, C, p)$
6 Compute $C_i = A_i \times B_i \mod p$ for $i = 0, 1, ..., n-1$.
7 Create array $W$ of inverse powers of $\omega$.
8 Compute fft1$(C, \omega^{-1}, n, W, T, p)$
9 Compute $C_i = n^{-1} \times C_i \mod p$ for $i = 0, 1, ..., n-1$.
```

Figure 2: Multiplying $a \times b$ using the FFT

Steps 1, 2 and 3 can be executed in parallel. Likewise steps 6 and 7 may be executed in parallel. Note, in step 7 the inverses can be computed from the original powers without any additional multiplications from $\omega^{-i} = -\omega^{\frac{n}{2}-i}$.

In step 9, multiplication by n^{-1} results in an additional pass through C. This pass can be eliminated by delaying multiplication of C_i by n^{-1} until Chinese remaindering.

We summarize by counting the number of multiplications $M(n)$ per prime for later reference. The three FFTs in steps 4,5 and 8 do $3F(n) = \frac{3}{2}n \log_2 n - \frac{3}{2}n$ multiplications. There are $\frac{n}{2} - 1$ multiplications in step 1 and n multiplications in step 6 so $M(n) = \frac{3}{2}n \log_2 n + O(1)$ multiplications.

3. TIME AND SPACE OPTIMIZATIONS

3.1 Optimization 1: Eliminating sorting

Figure 3 below is C code for a alternative radix 2 version of the FFT. The two algorithms `fft1` and `fft2` for the FFT presented in Figures 1 and 3 are known in the literature as the decimation in time (DIT) FFT and the decimation in frequency (DIF) FFT. Both compute $F = [a(1), a(\omega), a(\omega^2), \ldots, a(\omega^{(n-1)})]$, the Fourier transform of $a(x)$.

```
void fft2( int *A, int n, int *W, int p, int *T )
{  int i,n2,t;
   if( n==1 ) return;
   n2 = n/2;
   // Step 1 : arithmetic
   for( i=0; i<n2; i++ ) {
      T[i] = addmod(A[i],A[n2+i],p);
      t = submod(A[i],A[n2+i],p);
      T[n2+i] = mulmod(t,W[i],p); }
   // Step 2 : recursive calls
   fft2( T,    n2, W+n2, p, A    );
   fft2( T+n2, n2, W+n2, p, A+n2 );
   // Step 3 : permute
   for( i=0; i<n2; i++ ) {
      A[  2*i] = T[i];
      A[2*i+1] = T[n2+i]; }
   return;
}
```

Figure 3: Second FFT construction

To see how the two FFTs differ we first express $F = Va$ where $a = [a_0, a_1, \ldots, a_{n-1}]^T$ is the coefficient vector and V is the associated Vandermonde matrix. For $n = 4$ with

$\omega = i$ the matrix

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}.$$

Algorithm `fft1` first permutes the input $[a_0, a_1, a_2, a_3]$ to be $[a_0, a_2, a_1, a_3]$ using the permutation matrix

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

It then multiplies by A then by B where

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & i \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -i \end{bmatrix}.$$

So `fft1` uses the matrix factorization $V = BAP$. Algorithm `fft2` multiplies first by B^T then by A then P, thus using the matrix factorization $V = PAB^T$. Note $V^T = V$ and $(BAP)^T = P^T A^T B^T = PAB^T$.

If we are using the FFT to compute $c(x)$ the product of $a(x) \times b(x)$ in $\mathbb{Z}_p[x]$ with $\omega \in \mathbb{Z}_p$ then we will compute

$$c(x) = n^{-1}\, FFT(FFT(a, \omega, n) \times FFT(b, \omega, n), \omega^{-1}, n)$$

where we can use either `fft1` or `fft2` for the FFT here. If we do this as follows

$$c(x) = n^{-1}\, \texttt{fft1}(\texttt{fft2}(a, \omega, n) \times \texttt{fft2}(b, \omega, n), \omega^{-1}, n)$$

then since the permutation P satisfies $P^2 = I$, the permutation at end of `fft2` cancels out the permutation at the beginning of `fft1`. Therefore we can simply eliminate both permutation steps from both `fft1` and `fft2`. Moreover, we can eliminate T and save half the data movement and one third of the memory. The two radix 2 FFT codes become

```
void fft1( int *A, int n, int *W, int p )
{  int i,n2,s,t;
   if( n==1 ) return; else n2 = n/2;
   fft1( A,    n2, W+n2, p );
   fft1( A+n2, n2, W+n2, p );
   for( i=0; i<n2; i++ ) {
      s = A[i];
      t = mulmod(W[i],A[n2+i],p);
      A[   i] = addmod(s,t,p);
      A[n2+i] = submod(s,t,p); }
   return;
}
```

and

```
void fft2( int *A, int n, int *W, int p )
{  int i,n2,s,t;
   if( n==1 ) return; else n2 = n/2;
   for( i=0; i<n2; i++ ) {
      s = addmod(A[i],A[n2+i],p);
      t = submod(A[i],A[n2+i],p);
      A[   i] = s;
      A[n2+i] = mulmod(t,W[i],p); }
   fft2( A,    n2, W+n2, p );
   fft2( A+n2, n2, W+n2, p );
   return;
}
```

3.2 Optimization 2: Butterfly throughput

We have implemented a Radix-4 FFT for both versions of the FFT. A recursive Radix-4 FFT does four FFTs of size $n/4$. See Chapter 11 of [3] for details. The code for the DIF FFT is

```
void fft2( int *A, int n, int *W, int p )
{  int i,n2,n3,n4;
   if( n==1 ) return;
   if( n==2 ) { ... }
   n2 = n/2; n4 = n/4; n3 = n2+n4;
   for( i=0; i<n4; i++ ) {
      ... radix 4 butterfly ...
   }
   fft2( A,    n4, W+n4, p );
   fft2( A+n4, n4, W+n4, p );
   fft2( A+n2, n4, W+n4, p );
   fft2( A+n3, n4, W+n4, p );
   return;
}
```

where the radix 4 butterfly is given by

$$\begin{aligned} s_1 &= A_i + A_{i+\frac{n}{2}} & A_i &= s_1 + s_2 \\ s_2 &= A_{i+\frac{n}{4}} + A_{i+\frac{3n}{4}} & A_{i+\frac{n}{4}} &= (s_1 - s_2)\omega^{2i} \\ t_1 &= (A_i - A_{i+\frac{n}{2}})\omega^i & A_{i+\frac{n}{2}} &= t_1 + t_2 \\ t_2 &= (A_{i+\frac{n}{4}} - A_{i+\frac{3n}{4}})\omega^{i+\frac{n}{4}} & A_{i+\frac{3n}{4}} &= (t_1 - t_2)\omega^{2i} \end{aligned}$$

One advantage of the Radix-4 FFT is that we save half the passes through the data. A second advantage is that the radix 4 butterfly enables the compiler to improve throughput. In our implementation we have also unrolled the loop to count `i+=2` to further improve throughput. Also, we only need to store the following sequences of powers of ω in W

$$\begin{aligned} W = [\quad & \omega^i && \text{for } 0 \le i < n/2, \\ & \omega^{4i} && \text{for } 0 \le i < n/8, \\ & \omega^{16i} && \text{for } 0 \le i < n/16, \ldots]. \end{aligned}$$

Thus the memory needed for W is reduced from n to $\frac{2}{3}n$. The column labelled radix 2 in Table 2 shows the gain. We also tried a Radix-8 FFT. The gain was less than 2%.

Because $\omega^{\frac{n}{2}} = -1$ it follows that the quantity $j = \omega^{\frac{n}{4}}$ satisfies $j^2 = -1$ thus j is the complex unit. Therefore $\omega^{i+n/4} = \omega^i j$ and the radix 4 butterfly can be rearranged as follows so that there are three multiplications by powers of ω and one multiplication by j

$$\begin{aligned} s_1 &= A_i + A_{i+\frac{n}{2}} & A_i &= s_1 + s_2 \\ s_2 &= A_{i+\frac{n}{4}} + A_{i+\frac{3n}{4}} & A_{i+\frac{n}{4}} &= (s_1 - s_2)\omega^{2i} \\ t_1 &= A_i - A_{i+\frac{n}{2}} & A_{i+\frac{n}{2}} &= (t_1 + t_2)\omega^i \\ t_2 &= (A_{i+\frac{n}{4}} - A_{i+\frac{3n}{4}})\, j & A_{i+\frac{3n}{4}} &= (t_1 - t_2)\omega^{3i} \end{aligned}$$

In $\mathbb{C}$, multiplication by j is free so this saves one quarter of the complex multiplications. This makes the Radix-4 FFT popular for floating point FFT implementations. However, in $\mathbb{Z}_p$, multiplication by $j = \omega^{n/4}$ is no different than multiplication by other powers of ω so nothing is gained. Are there any arithmetic optimizations in the Radix-4 butterfly available to $\mathbb{Z}_p$? In the four subtractions of the form $(a-b)\omega^k$ the difference $a-b$ can be computed without reduction mod p in an long accumulator using `((long) a+p-b)*W[k] % p`. This saves an additional 9% of the total FFT time for 31 bit primes.

3.3 Optimization 3: Chinese remaindering

After using the FFT to multiply $c_i(x) := a(x) \times b(x) \bmod p_i$ for m primes $p_1, p_2, \ldots, p_m$ we must solve the Chinese remainder problem

$$\{ \ \bar{c}(x) \equiv c_i(x) \mod p_i \ \text{ for } 1 \le i \le m \ \}$$

for $\bar{c}(x)$ then we reduce the integer coefficients of $\bar{c}(x)$ mod q to obtain $c(x)$. Although the Chinese remaindering costs $O(m^2 n)$ and the $3m$ FFTs cost $O(mn \log n)$ and m is small, the cost of the Chinese remaindering is significant even for large n, so it needs to be done carefully.

Let $M = p_1 \times p_2 \times \cdots \times p_m$. The Chinese remainder theorem says, given integers $u_1, u_2, \ldots, u_m$ there exists a unique integer $0 \le u < M$ satisfying $u \equiv u_i \mod p_i$.

To compute u many authors, e.g., von zur Gathen and Gerhard in [6], use the following method. Let

$$w = w_1 \frac{M}{p_1} + w_2 \frac{M}{p_2} + \cdots + w_m \frac{M}{p_m}. \quad (1)$$

If one reduces (1) modulo p_i all the terms on the right-hand-side except one vanish so that if we solve $u_i \equiv w_i(M/p_i) \bmod p_i$ for w_i then the integer w will satisfy $w \equiv u_i \mod p_i$. What is attractive about this approach is that we can pre-compute the inverses $(M/p_i)^{-1} \bmod p_i$. Then computing $w_i = u_i(M/p_i)^{-1} \bmod p_i$ requires one multiplication and one division by p_i. And if it were the case that $w = u$ if we also precompute $(M/p_i) \mod q$ then we could compute w mod q using (1) in an accumulator using m multiplications and only one division by q. Unfortunately w may not be equal to u, that is, $w = u + \alpha M$ for some $\alpha \ge 0$. How big can w be? We have $0 \le w_i < p_i$ therefore

$$\begin{aligned} w &\le (p_1-1)\frac{M}{p_1} + (p_2-1)\frac{M}{p_2} + \cdots + (p_m-1)\frac{M}{p_m} \\ &= mM - \frac{M}{p_1} - \frac{M}{p_2} - \cdots - \frac{M}{p_m}. \end{aligned}$$

This maximum for w is attained when $u_i = -(M/p_i) \mod p_i$. For in this case we have

$$w_i = u_i \left(\frac{M}{p_i}\right)^{-1} = -\frac{M}{p_i}\left(\frac{M}{p_i}\right)^{-1} \equiv -1 \bmod p_i.$$

Therefore w approaches $m \times M$. Thus to obtain $u \mod q$, one must first construct w, which requires multi-precision arithmetic in general, then reduce w mod M to get u then reduce u mod q.

We advocate instead for the mixed radix (Newton) representation. Let

$$v = v_1 + v_2 p_1 + v_3 p_1 p_2 + \cdots + v_m p_1 p_2 \ldots p_{m-1}. \quad (2)$$

We solve (2) modulo $p_1, p_2, \ldots, p_m$ for $v_1, v_2, \ldots, v_m$ in that order to get, for example,

$$v_3 = (u_3 - v_1 - v_2 p_1)\,(p_1 p_2)^{-1} \mod p_3.$$

Since $0 \le v_i < p_i$ the maximum value for v occurs when $v_i = p_i - 1$ from which we have

$$v \le (p_1-1) + (p_2-1)p_1 + \cdots + (p_m-1)p_1 p_2 \ldots p_{m-1}. \quad (3)$$

The right hand side of (3) collapses to $M-1$ therefore $v = u$. Furthermore, since our goal is to compute $u \mod q$, we do not need to explicitly construct v. If we pre-compute $p_1 p_2$ mod q we can compute $v_0 + v_1 p_1 + v_2 p_1 p_2 \mod q$ using an accumulator with one division by q instead of two. Further optimizations are possible. Assuming in Figure 2 that we did not multiply C by n^{-1} then we must do that here. We illustrate this for $m = 3$ primes p_1, p_2, p_3.

We pre-compute $n_1 = n^{-1} \bmod p_1$, $n_2 = n^{-1} \mod p_2$, $z_1 = p_1^{-1} \mod p_2$, $z_2 = (p_1 p_2)^{-1} \mod p_3$, and $z_3 = p_1 p_2$ mod q. Now we can compute $u \mod q$ using an accumulator as follows using 8 multiplications and 6 divisions per coefficient of $c(x)$.

```
1 $u_1 = n_1 \times u_1 \mod p_1$
2 $u_2 = n_2 \times u_2 \mod p_2$
3 $v_1 = u_1$
4 $v_2 = (u_2 - v_1) \times z_1 \mod p_2$
5 $t = (n_3 \times u_3 - v_1 - v_2 \times p_1) \mod p_3$
6 $v_3 = t \times z_2 \mod p_3$
7 $u = (v_1 + v_2 \times p_1 + v_3 \times z_3) \mod q$
```

3.4 Optimization 4: W parallelization

Recall that the total number of multiplications done per prime is $M(n) = \frac{3}{2} n \log_2 n + O(1)$. Although computing the powers $1, \omega, \omega^2, \ldots, \omega^{n/2-1}$ requires only $\frac{n}{2} - 2$ multiplications which seems few compared with $M(n)$, even at degree 10^6, the time spent doing them is enough to significantly limit parallel speedup. Amdahl's law says that on a machine with N cores, parallel speedup Q is bounded by

$$Q \le \frac{P+S}{P/N+S} \quad (4)$$

where P is work done in parallel and S is the sequential work. If we compute W sequentially and everything else in parallel, then $S = \frac{n}{2}$ and $P = M(n) - \frac{n}{2}$. Evaluating the right-hand-side of (5) for $N = 16$ cores, we calculate parallel speedup $Q \le 12.8$ for $n = 2^{20}$ and $Q \le 13.7$ for $n = 2^{30}$.

For $n > 2^{16}$ we compute W a block at a time with block-size $m = 2^{16}$. We compute $\beta = \omega^m$ using repeated squaring modulo p then we compute blocks

$$[\beta^i, \omega\beta^i, \omega^2\beta^i, \ldots, \omega^{m-1}\beta^i]$$

for $0 \le i < n/m$ in parallel by calling the routine below with input $m, \omega, \omega^4, \beta^i$ and pointer $W + mi$. To improve throughput we have unrolled the loop.

```
void Wpowersp1( int m, long w, long w4,
                int betai, int *W )
{  int i,p;
   p = 2013265921; // p1 = 15 x 2^27 + 1
   W[0] = betai;        W[1] = w*W[0] % p;
   W[2] = w*W[1] % p;   W[3] = w*W[2] % p;
   for( i=4; i<m; i+= 4 ) {
      W[i  ] = w4*W[i-4] % p;
      W[i+1] = w4*W[i-3] % p;
      W[i+2] = w4*W[i-2] % p;
      W[i+3] = w4*W[i-1] % p;
   return;
}
```

4. BENCHMARKS

We used an Intel server with 128 gigabytes of RAM running RedHat Linux. It has two Intel Xeon E5-2680 v2 processors. Each has 10 cores and runs at 2.8 GHz (3.6 GHz turbo). Thus the maximum parallel speedup is $20 \times 2.8/3.6 = 15.55\times$. The maximum throughput of our assembler code for multiplication modulo a 63 bit prime using the algorithm from [15] is 5.28 clock cycles.

We multiplied two polynomials of degree 1 billion in $\mathbb{Z}_q[x]$ for $q = 2^{31} - 1$ using two 63 bit primes. The space needed for the two inputs and output 32 gigabytes and the working space used by our algorithm is 85.3 gigabytes. The time on one core was 1376.86 seconds and 82.79 seconds on 20 cores yielding a parallel speedup of a factor of 16.6, more than the theoretical maximum (linear speedup) which happens because of improved locality. The throughput achieved by our FFT code on 20 cores is $\frac{82.79 \times 3.0\, 10^9 \times 20 cores}{2M(n)} = 24.9$ clock cycles per core for one multiplication, one division, one addition and one subtraction. Note $2M(n)$ is the number of multiplications (see Section 2.4) for the two 63 bit prime modular multiplications.

Table 1 shows the parallel speedup achieved by our implementation for multiplying polynomials $\mathbb{Z}_q[x]$ for $q = 2^{31} - 1$ and how our code compares with Maple 18 and Magma 2.13 which are using the Kronecker substitution (Section 1). Column 1 is the degree (in millions) of the input polynomials $a(x)$ and $b(x)$. Column 2 and 3 are the time to compute a^2 and $a \times b$ using a serial code (no Cilk). The next columns use our parallel Cilk code for 1,3,6,10,15 and 20 cores. The data shows that the Cilk overhead on 1 core is not negligible. The timings for Maple 18 and Magma 2.13 are similar because they both use GMP to multiply long integers. The code we used to generate and multiply the polynomials in $\mathbb{Z}_q[x]$ in Maple and Magma is given in the Appendix.

That data shows that our serial code is about 3 times faster than the Kronecker substitution using GMP 5.0's integer multiplication in Maple and Magma. It means that our 3 prime FFT implementation is 3 times faster than the GMP integer multiplication which we think is a very good result. That data also shows that we get good parallel speedup at degree 1 million. We are presently trying to tune the various cutoffs so that we improve parallel speedup for lower degree without sacrificing speed at higher degree.

Table 2 shows the effect of some of optimizations to the serial code for multiplying $a \times b$ in $\mathbb{Z}_q[x]$ for $q = 2^{31} - 1$ using the modular method. Column 1 is the degree (in millions) of the input polynomials. Columns 2 and 8 (opt) are timings for fully optimized codes for three 31 bit primes and two 63 bit primes, respectively. Column 3 (+div) uses hardware division to reduce mod p (see Section 2.3). Columns 4 and 9 (add/sub) are timings with addition and subtraction using an if statement (see Section 2.3). Columns 5 and 10 (W stride) show the disastrous effect of cache misses if we do not do the optimization in Section 2.1 for W. Columns 6 and 11 (+sort) is without the optimization in Section 3.1. Columns 7 and 12 (radix 2) show what happens if we use simple Radix-2 FFT codes instead of Radix-4 FFT codes with loop unrolling (see Section 3.2). All of the optimizations clearly matter if we want good serial code.

Tables 3 and 4 show the impact of reducing parallelism. In both tables the first column is the degree (in millions) of the input polynomials a and b. The second and third columns are the times (in seconds) for multiplying with all parallelism on for 1 core and 20 cores. The fourth column is the parallel speedup (the time in column 2 divided by the time in column 3). The timings in column $-ppar$ are for using 20 cores but doing each prime sequentially instead of in parallel. This shows how good the the FFT is parallelized. The timings in column $-wpar$ are for 20 cores with W constructed sequentially.

Finally, Table 5 shows the improvement on typical desktop processors at degree 1 million.

5. CONCLUSION AND CURRENT WORK

We have developed serial code for multiplying polynomials in $\mathbb{Z}_q[x]$ which uses Chinese remaindering. It multiplies modulo several primes $p_1, p_2, \ldots, p_m$ where multiplication modulo each prime uses the FFT. We have parallelized the code on the primes and we have parallelized the FFT. We get good parallel speedup at degree one million on multicore platforms. We mention two applications where such high degree multiplications naturally arise.

The first is to multiply two large integers $a \times b$. If B is the base of the integer representation (e.g. GMP uses $B = 2^{64}$), and $a = f(B)$ where $f(x) = \sum_{i=0}^{na} a_i x^i$ and $b = g(B)$ where $g(x) = \sum_{i=0}^{nb} b_i x^i$ then we can obtain $c = a \times b$ by first multiplying $h(x) = f(x) \times g(x)$ in $\mathbb{Z}_B[x]$ using our parallel algorithm then evaluating $h(x)$ at $x = B$ to get c.

Another application is sparse polynomial interpolation when the number of terms t of the polynomial is very large. The algorithm of Hu and Monagan [10] computes the support of the polynomial by picking a prime q of a certain form. The most expensive step of the algorithm requires computation of the roots of a polynomial $\lambda(z) \in \mathbb{Z}_q[z]$ of degree t. For large t fast multiplication in $\mathbb{Z}_q[z]$ can be used to accelerate this step. See see Algorithm 14.15 of [6] and Shoup [20]. The second author is presently implementing these algorithms.

6. REFERENCES

[1] David Bailey, FFTs in external or hierarchical memory. *J. Supercomputing*, **4**(1) pp. 23–35, 1990.

[2] A. Chmielowiec. Fast, Paralelel Algorithm for Multiplying Polynomials In *IAENG Transactions on Engineering Technologies*, Springer Verlag LNEE **229**, pp. 605–616, 2012.

[3] E. Chu and A. George. *Inside the FFT Black Box* CRC Press, Computational Mathematics Series, 2000.

[4] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, J. Moura. Discrete Fourier Transform on Multicores. IEEE Signal Processing Magazine, **26**(6) pp. 90-102, November 2009.

[5] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, **93** (2) pp. 216–231, 2005. See also http://www.fftw.org/

[6] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, 3rd ed., Cambridge University Press, 2013.

[7] The GNU Multiple Precision Arithmetic Library. `https://gmplib.org/`

[8] David Harvey. A cache-friendly truncated FFT *J. Theoretical Computer Science*, **410**, 2649–2658, 2009.

[9] J. van der Hoeven. The Truncated Fourier Transform and Applications. *Proceedings of ISSAC 2004*, ACM Press, pp. 290-296, 2004.

Table 1: Real time (in seconds) for multiplying $a \times b$ modulo $q = 2^{31} - 1$ using three 31 bit primes.

	Serial timings		Parallel timings for $a \times b$ using n cores							Maple		Magma	
Deg	a^2	$a \times b$	1	3	6	10	15	20	speedup	a^2	$a \times b$	a^2	$a \times b$
0.25m	0.085	0.117	0.127	0.044	0.024	0.016	0.012	0.012	10.6x	0.26	0.34	0.22	0.32
0.5m	0.182	0.254	0.273	0.095	0.053	0.036	0.026	0.024	11.4x	0.52	0.72	0.52	0.65
1m	0.367	0.515	0.566	0.195	0.102	0.067	0.048	0.038	14.9x	1.08	1.56	1.08	1.50
2m	0.787	1.108	1.206	0.415	0.220	0.142	0.102	0.081	14.9x	2.21	3.36	2.38	2.94
4m	1.595	2.244	2.485	0.853	0.448	0.291	0.205	0.160	15.5x	4.67	6.86	6.28	7.89
8m	3.407	4.802	5.280	1.812	0.945	0.612	0.430	0.335	15.8x	9.84	14.49	11.6	14.8
16m	7.089	10.017	11.19	3.841	2.015	1.296	0.905	0.699	16.0x	21.79	32.09	26.9	34.2
32m	15.056	21.309	23.61	8.107	4.234	2.741	1.898	1.462	16.2x	48.36	71.34	50.5	69.3

Table 2: Serial timings in seconds for $q = 2^{31} - 1$ for some optimizations turned off.

Deg	using three 31 bit primes						using two 63 bit primes				
(million)	opt	+ div	add/sub	W stride	+ sort	radix 2	opt	add/sub	W stride	+ sort	radix 2
1m	0.516	1.857	0.951	0.737	0.715	1.494	0.810	1.404	1.043	1.009	1.103
2m	1.116	3.962	2.000	1.582	1.579	3.189	1.712	2.959	2.208	2.168	2.347
4m	2.257	8.147	4.131	3.278	3.208	6.767	3.535	6.141	8.455	4.538	4.978
8m	4.828	17.293	8.657	15.715	7.158	14.296	7.589	12.993	20.014	10.072	10.513
16m	10.076	35.628	18.045	38.491	15.535	30.114	15.635	26.851	45.943	20.839	22.132
32m	21.396	75.246	37.661	88.185	33.965	63.314	33.211	56.408	101.45	45.490	46.496

Table 3: Parallel timings (real times in seconds) for three 31 bit primes.

Deg	1 core	20 cores	speedup	-ppar	speedup	-wpar	speedup
.25m	0.126	0.014	9.0x	0.025	5.04x	0.014	9.0x
0.5m	0.272	0.025	10.9x	0.053	5.13x	0.026	10.5x
1m	0.561	0.039	14.4x	0.047	11.9x	0.040	14.0x
2m	1.200	0.084	14.3x	0.097	12.4x	0.084	14.3x
4m	2.475	0.169	14.6x	0.184	13.5x	0.163	15.2x
8m	5.254	0.344	15.3x	0.384	13.7x	0.343	15.3x
16m	11.13	0.702	15.9x	0.768	14.5x	0.703	15.8x
32m	23.49	1.464	16.0x	1.599	14.7x	1.475	15.9x

Table 4: Parallel timings (real times in seconds) for two 63 bit primes. −ppar = no parallelization on the primes. −wpar = no parallelization of W.

Deg	1 core	20 cores	speedup	-ppar	speedup	-wpar	speedup	Maple	Magma
1m	0.859	0.065	13.2x	0.069	12.4x	0.066	13.0x	1.56	1.50
2m	1.822	0.138	13.2x	0.149	12.2x	0.168	10.8x	3.36	2.94
4m	3.751	0.259	14.5x	0.279	13.4x	0.277	13.5x	6.86	7.89
8m	8.012	0.547	14.6x	0.605	13.2x	0.599	13.4x	14.49	14.80
16m	16.448	1.052	15.6x	1.185	13.9x	1.177	14.0x	32.09	34.20
32m	34.961	2.332	15.0x	2.507	13.9x	2.413	14.9x	71.34	69.30
64m	71.727	4.698	15.3x	5.108	14.0x	4.989	14.4x	170.4	170.7
128m	152.28	9.888	15.4x	10.63	14.3x	10.44	15.6x	348.6	410.5
256m	313.28	19.751	15.9x	21.61	14.5x	21.21	14.8x	766.2	>MEM
512m	661.70	42.666	15.5x	45.76	14.5x	44.92	14.7x	>MEM	–
1024m	1375.5	84.487	16.3x	93.45	14.7x	91.22	15.1x	–	–

Table 5: Real times to multiply two polynomials of degree 1 million mod $q = 2^{31} - 1$

	#cores	clock (GHz)	Maple	modular method		speedup	
CPU	n	base/turbo	Kronecker	1 core	n cores	parallel	overall
Core i7 920	4	2.7/2.9	2.860s	1.080s	0.310s	3.5x	9.2x
Core i7 2600	4	3.4/3.8	1.517s	0.638s	0.180s	3.5x	8.4x
AMD FX 8350	8	4.0/4.2	2.070s	1.012s	0.163s	6.2x	12.7x
Core i5 4570	4	3.2/3.6	1.406s	0.493s	0.135s	3.7x	10.4x
Core i7 3930K	6	3.2/3.8	1.407s	0.582s	0.106s	5.5x	13.2x
2x Xeon E5 2660	16	2.2/3.0	1.845s	0.788s	0.061s	12.9x	30.2x
2x Xeon E5 2680v2	20	2.8/3.6	1.574s	0.540s	0.038s	14.2x	41.4x

[10] J. Hu and M. Monagan. A Parallel Algorithm to Compute the Greatest Common Divisor of Sparse Multivariate Polynomials. *Communications in Computer Algebra 47:3*, pp. 108–109, 2013.

[11] J. Johnson, R. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circuit, Systems, and Signal Processing*, **9**(4) pp. 249–500, 1990.

[12] X. Lik, M. Moreno Maza, R. Rasheed, E. Schost. The `modpn` library: Bringing fast polynomial arithmetic into maple. *J. Symb. Comp.* **46**(7) pp. 841–858, 2011.

[13] L. Meng and J. Johnson. High Performance Implementation of the TFT. *Proceedings of ISSAC 2014*, ACM Press, pp. 328–334, 2014.

[14] M. Moreno Maza and Y. Xie. FFT-based Dense Polynomial Arithmetic on Multi-cores In High Performance Computing Systems and Applications, Springer-Verlag LNCS **5976**, pp 378–399, 2009.

[15] Niels Möller and Torbjorn Granlund. Improved division by invariant integers IEEE Transactions on Computers, **60**, 165–175, 2011.

[16] M. B. Monagan. In-place arithmetic for polynomials over $\mathbf{Z}_n$. *Proceedings of DISCO '92*, Springer-Verlag LNCS, **721**, pp. 22–34, 1993.

[17] M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, **93** (2) pp. 232–275, 2005.

[18] A. Schönhage and V. Strassen. Schnelle Multiplikation groβer Zahlen. *Computing* **7** pp. 281–292, 1971.

[19] V. Shoup. Factoring Polynomials over Finite Fields: Asymptotic Complexity vs. Reality. *Proceedings of IMACS Symp. on Symb. Cmpt.*, Lille, France, pp. 124–129, 1993.

[20] V. Shoup. A New Polynomial Factorization Algorithm and its Implementation. *J. Symb. Comp.* **20** pp. 363–395, 1995.

[21] Victor Shoup. NTL: A library for doing number theory. `http://www.shoup.net/ntl/`

Appendix

Maple code used for timing multiplication in $\mathbb{Z}_q[x]$.

```
q := 2147483647;
d := 10^6;
A := Randpoly(d,x) mod q:
B := Randpoly(d,x) mod q:
time[real]( Expand(A*A) mod q );
time[real]( Expand(A*B) mod q );
```

Magma code used for timing multiplication in $\mathbb{Z}_q[x]$.

```
q := 2147483647;
Fq := FiniteField(q);
P<x> := PolynomialRing(Fq);
d := 10^6;
f := P ! [ Random(Fq) : s in [0..d] ];
g := P ! [ Random(Fq) : s in [0..d] ];
time s := f*f;
time h := f*g;
```