Parallelized Polynomial Multiplication (MultPoly)

Yaxin Chen (yc3995)

December 2021

1 Introduction

In this project, I implemented three algorithms for polynomial multiplication in Haskell. The first one is the native approach with a time complexity of $O(n^2)$, where n is the degree of the polynomial; the second one utilizes recursive fast fourier transform (FFT) and has a time complexity of $O(n \log n)$; the third one also utilizes Cooley-Tukey (CT) algorithm, which is a iterative algorithm for FFT, and also has a time complexity of $O(n \log n)$. Testing with 4 cores and polynomials of length 10000, the parallel brute force approach achieved a 5.87x speedup, the parallel FFT achieved a 1.78x speedup, and the parallel CT achieved a 1.73x speedup on Intel i7-10750H CPU.

2 Implementation

A degree-(n-1) polynomial can be represented by an n-element array storing its coefficients. For implementation, I use Haskell List to store these coefficients.

2.1 Brute Force

2.1.1 Sequential Solution (BF)

Suppose we have array A representing polynomial $a(x) = \sum_{i=0}^{n-1} A[i]x^i$ and B representing polynomial $b(x) = \sum_{i=0}^{n-1} B[i]x^i$, the array C for the product of $a(x)$ and $b(x)$ can be calculated by Alg. 1.

Algorithm 1: Brute Force Approach for Polynomial Multiplication

```
1 for i ← 0 to n-1 do
2 |   for j ← 0 to n-1 do
3 |   |   C[i + j] ← C[i + j] + A[i] * B[j];
4 |   end for
5 end for
```

I implemented two parallel solutions for brute force approach, using divide and conquer and map reduce framework respectively.

2.1.2 Parallel Solution with Divide and Conquer (BFPAR)

Similar to the logic of integer multiplication, multiplying two polynomials A, B is equivalent to multiplying each coefficient of A with B and then shift and add them together. Therefore, we can break A into two parts, $A_1 = A[0..(n/2-1)]$ and $A_2 = A[n/2..(n-1)]$, and $C = (A_1 * B) + ((A_2 * B) << (n/2))$, where $X << l$ means append an zero array of length l to X. The calculation of $A_1 * B$ and $A_2 * B$ can be parallelized. The corresponding algorithm is shown in Alg. 2

Algorithm 2: Parallel Brute Force Approach

```
   Input  : two polynomials A and B, parallel depth d
   Output: polynomial C = A * B
 1 Function ParBF(A, B, d):
 2     l_A ← lengthA;
 3     l_B ← lengthB;
 4     if d <= 0 or l_A = 1 then
 5         for i ← 0 to l_A-1 do
 6             for j ← 0 to l_B-1 do
 7                 C[i + j] ← C[i + j] + A[i] * B[j];
 8                 return C;
 9             end for
10         end for
11     end if
12     C_1 = ParBF(A[0..(l_A/2 − 1)], B, d − 1);
13     C_2 = ParBF(A[l_A/2..(l_A − 1)], B, d − 1);
14     C = C_1 + C_2 << (l_A/2);
15     return C;
16 end
```

2.1.3 Parallel Solution with Map Reduce (BFMP)

Observing that for any k, q $\in 0, 1, ..., lengthC$, the formulation of $C[k]$ is independent of that of $C[j]$, I considered using map reduce framework to solve this problem. The mapper takes pairs of coefficient of two input polynomials (A[i], B[j]), multiplies them, and sends (key: i+j, value: A[i]*B[j]) to the reducer. The reducer sums the received products and gives output coefficient at index (i+j). To send the mapper result to the reducer, there is a shuffle function that gathers all mapper results and sort them according to the key.

This approach failed miserably, which is analyzed in Experiments section.

2.2 Recursive Fast Fourier Transform

2.2.1 Sequential Solution (FFT)

The polynomial multiplication can be speed up to $O(n \log n)$ by fast fourier transforming the input polynomials, multiplying them and the inverse fourier transforming the product.

The discrete fourier transform (DFT) of an n-element sequence A is another n-element sequence P given by

$$P[m] = \sum_{k=0}^{n-1} A[k]\omega_n^{mk},\ m = 0, 1, ..., n-1$$

where $\omega_n = e^{2\pi i/n}$ is the primitive n^{th} root of unity.

For $0 \leq m < n/2$, DFT satisfies

$$P[m] = P_1[m] + \omega^m P_2[m] \quad (1)$$

$$P[n/2 + m] = P_1[m] - \omega^m P_2[m] \quad (2)$$

where

$$P_1[m] = \sum_{k=0}^{n/2-1} A[2k]\omega_n^{2mk}$$

$$P_2[m] = \sum_{k=0}^{n/2-1} A[2k+1]\omega_n^{2mk}$$

FFT utilizes the above property (Eq 1, 2). The algorithm for FFT is shown in Algorithm 3.

Algorithm 3: Fast Fourier Transform

Input : array A of length n, n^th root of unity ω
Output: Fourier transform of A

```
1  Function FFT(A, n, ω):
2      if n = 1 then
3          return A;
4      end if
5      else
6          for k ← 0 to n/2 - 1 do
7              A_1[k] = A[2k];
8              A_2[k] = A[2k + 1];
9          end for
10         P_1 ← FFT(A_1, n/2, ω^2);
11         P_2 ← FFT(A_2, n/2, ω^2);
12         // combine P_1 and P_2
13         for m ← 0 to n - 1 do
14             P[m] ← P_1[m mod (n/2)] + ω^m P_2[m mod (n/2)];
15         end for
16     end if
17 end
```

Since FFT can only be applied on array with length equal to some power of 2, we need to extend both polynomials to have length equal to the closest power of 2 by appending 0, i.e. $a(x) = \sum_{i=0}^{l_A-1} A[i]x^i = \sum_{i=0}^{l_A-1} A[i]x^i + \sum_{i=l_A}^{n} 0x^i$.

Algorithm 4: Polynomial Multiplication via FFT

Input : Two polynomials A and B of degree smaller then n/2, with n a power of 2, primitive nth root of unity ω
Output: C = AB

```
1  Expand A, B to have length n;
2  P ← FFT(A, ω);
3  Q ← FFT(B, ω);
4  S ← PQ;
5  C ← (1/n) FFT(S, ω^-1);
6  return C;
```

2.2.2 Parallel Solution (FFTPAR)

The parallel solution is done by parallelizing the fast fourier transform algorithm. In Alg. 3, the calculation of P_1 and P_2 are independent of each other, which can be parallelized.

```
fft l n w d = runEval $ do
  p1 <- rpar (force (fft l1 (n `div` 2) (w ** 2) (d - 1)))
  p2 <- rpar (force (fft l2 (n `div` 2) (w ** 2) (d - 1)))
  _ <- rseq p1
  _ <- rseq p2
  return $ combine p1 p2 w
  where
    (l1, l2) = split l
```

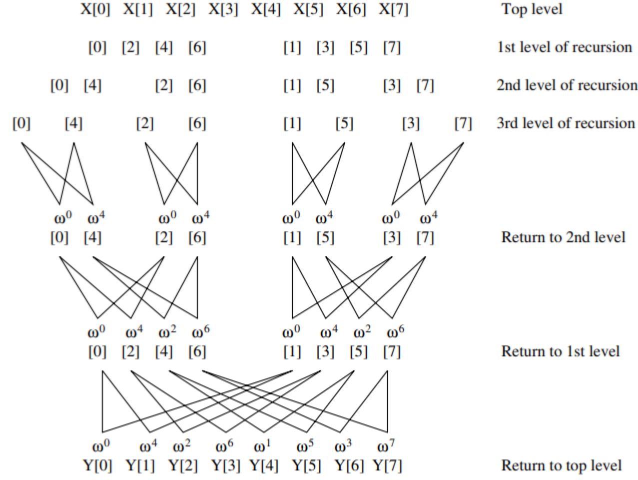

Figure 1: 8-point FFT

2.3 Iterative Fast Fourier Transform (Cooley-Tukey Algorithm)

2.3.1 Sequential Solution (FFTCT)

Fig. 1 and Fig. 2 shows the computational graph of Cooley-Tukey algorithm [5]. Before multiplying with ω, it needs to re-order the input. This process is called bit-reverse, which can be done in $O(n)$ with Array but $O(nlogn)$ with List. Then at each stage m, where $1 \leq m \leq logn$, the list breaks into the $n/2^m$ parts; each part l is transformed into l_{new}, where

$$l_{new}[i] = l[i\, mod\, (n/2)] + \omega^i l[i\, mod\, (n/2) + n/2], 0 \leq i \leq 2^m \tag{3}$$

2.3.2 Parallel Solution (FFTCTPAR)

Fig. 2 shows the computational graph of parallel Cooley-Tukey algorithm (without bit-reverse part), which is also called Binary Exchange algorithm [5]. Suppose there are 2^d threads and the calculation of FFT is distributed equally to these threads ($P_0 ... P_3$) as shown in Fig. 2.

During the first $\log n - d$ stages, there is no interaction among different threads. Therefore, I split the whole list of polynomial into 2^d lists, and use *parMap* to calculate the result of each list after the first $\log n - d$ stages, and concatenate 2^d result lists into one. For the last d - 1 stages, I use *parListChunk* to parallel calculation of Eq. 3.

3 Experiments

The experiments are performed on Ubuntu virtual machine with 2.60GHz 4-core Intel(R) Core(TM) i7-10750H CPU and 4GB memory.

3.1 Brute Force

3.1.1 Parallel Solution with Divide and Conquer

Table 1 shows the speed up of the parallel brute force approach with divide and conquer. The leftmost column is the length of both polynomials. The maximum speedup reached 6.93, which is even larger than the number of cores. A possible reason is that divide and conquer decreases the size of memory used from $O(n^2)$ to $O((\frac{n}{2})^2)$, which reduces the IO time.

Figure 3 shows a satisfying threadscope analysis of the parallel brute force approach with divide and conquer.

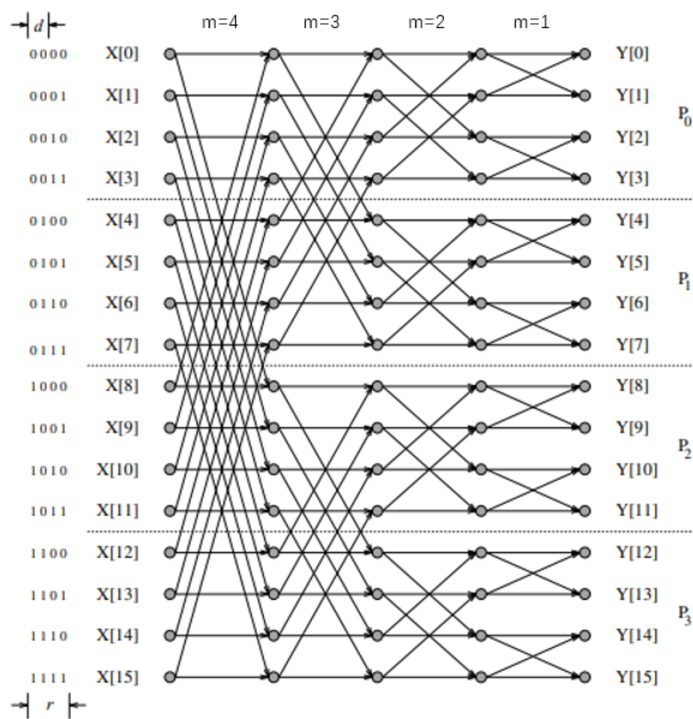

Figure 2: 16-point FFT on four processes

length	Brute Force	Parallel Divide and Conquer	Speedup
1000	0.284	0.252	1.126984
2000	0.605	0.287	2.108014
3000	1.15	0.383	3.002611
4000	2.151	0.542	3.968635
5000	2.89	0.74	3.905405
6000	4.56	1.026	4.444444
7000	7.41	1.45	5.110345
8000	11.32	1.848	6.125541
9000	15.52	2.239	6.931666
10000	19.82	3.378	5.867377

Table 1: Runtime of Brute Force Approach with Divide and Conquer.

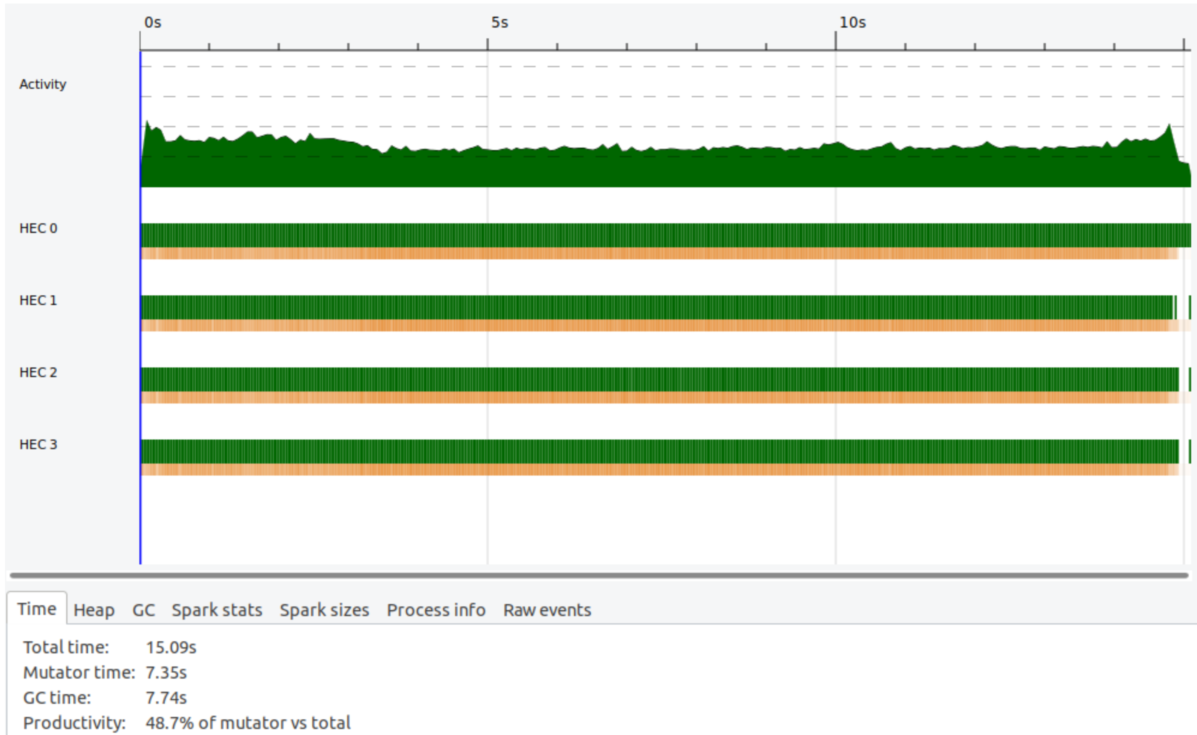

Figure 3: Threadscope of Brute Force Approach with Divide and Conquer (length of polynomials: 20000).

3.1.2 Parallel Solution with Map Reduce

Table 2 shows the runtime of the brute force approach paralleled with Map Reduce. For small input size, it takes longer time than the non-parallel version, and it even gets killed when the length of polynomial reaches 4000. Analyzed in ThreadScope (Figure 4), we can see that there is a long time that only one core works in the middle, which is taken up by the shuffle function. For polynomial multiplication, the calculation done by each mapper and reducer is little, while the shuffle function takes a lot of time and memory to sort all key-value pairs on a single machine, which also indicates that Map Reduce suits better for distributed system [7].

length	Brute Force	Map Reduce
1000	0.284	1.087
2000	0.605	3.512
3000	1.15	8.357
4000	2.151	killed

Table 2: Runtime of Brute Force Approach with Map Reduce.

3.2 Parallel Recursive Fast Fourier Transform

Table 3 shows the runtime of parallel recursive fast fourier transform. The speedup is around 2x. As shown in Figure 5), there is overhead in the parallel implementation at both the beginning and the end, which is due to $O(n)$ list split at the beginning and recombination at the end.

One interesting thing about FFT is that its result was incorrect at first and it turned out to be a precision problem. Changing Float to Double solves this problem, which indicates FFT requires higher precision than the brute force approach.

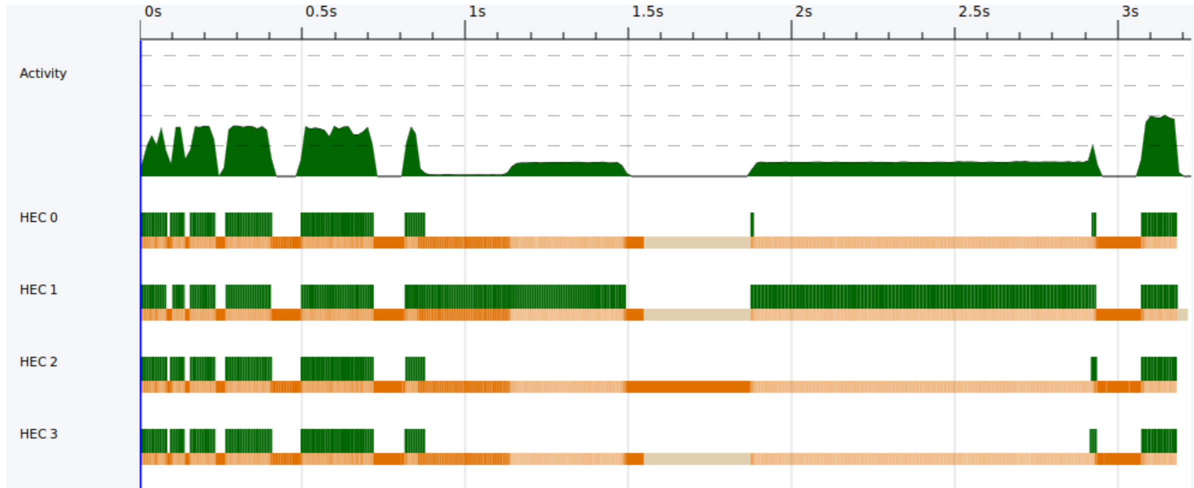

Figure 4: Threadscope of Brute Force Approach with Map Reduce(length of polynomials: 2000).

length	FFT	Parallel FFT	Speedup
10000	1.016	0.571	1.779335
20000	2.1	1.048	2.003817
30000	2.265	1.161	1.950904
40000	4.079	1.932	2.111284
50000	4.296	2.102	2.043768
60000	4.528	2.185	2.072311
70000	8.469	3.7729	2.244692
80000	8.536	3.928	2.173116

Table 3: Runtime of Parallel Recursive Fast Fourier Transform.

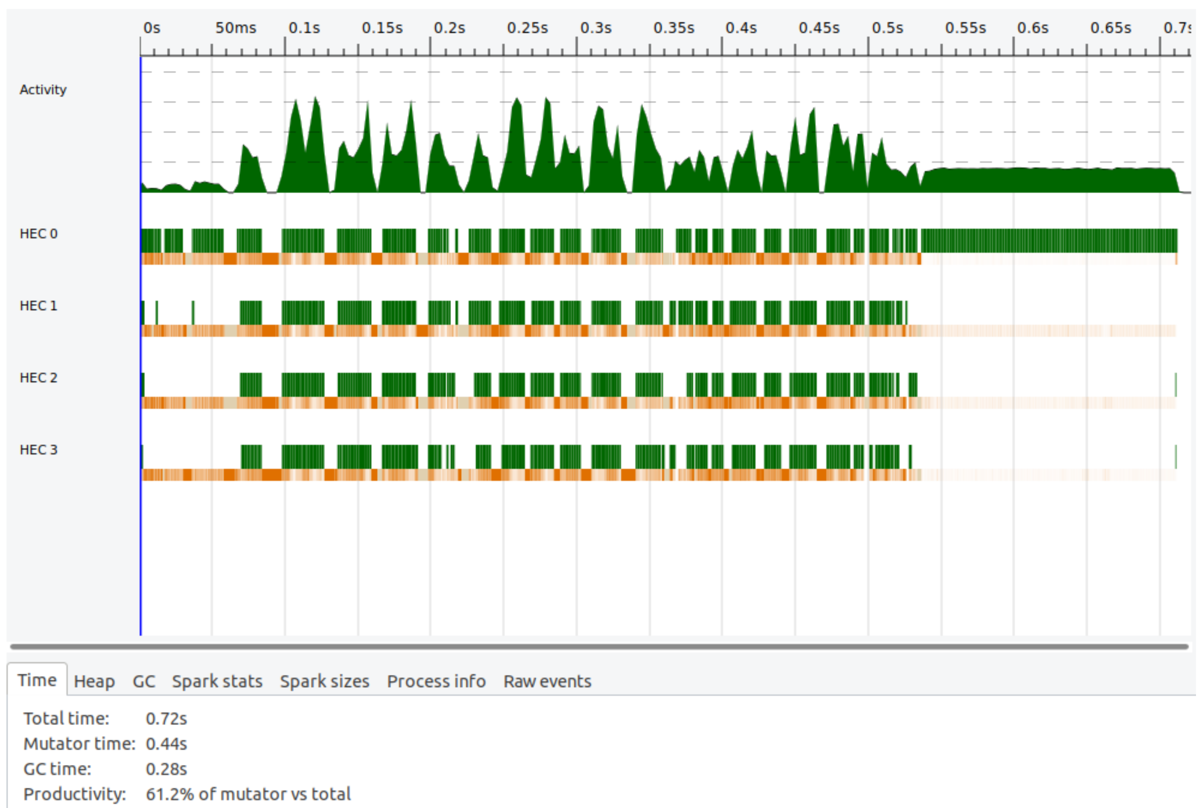

Figure 5: Threadscope of Parallel Recursive Fast Fourier Transform(length of polynomials: 20000).

3.3 Parallel Iterative Fast Fourier Transform (Cooley-Tukey)

Table 4 shows the runtime of parallel Cooley-Tukey algorithm. The speedup is also around 2x. The threadscope image of it (Figure 6) is similar to that of parallel recursive FFT.

length	Cooley-Tukey	Parallel Cooley-Tukey	Speedup
10000	1.339	0.773	1.732212
20000	2.644	1.412	1.872521
30000	2.789	1.613	1.729076
40000	5.288	2.724	1.941263
50000	5.416	2.866	1.889742
60000	5.641	3.107	1.815578
70000	10.585	5.342	1.981468
80000	11.028	5.421	2.034311

Table 4: Runtime of Parallel Iterative Fast Fourier Transform.

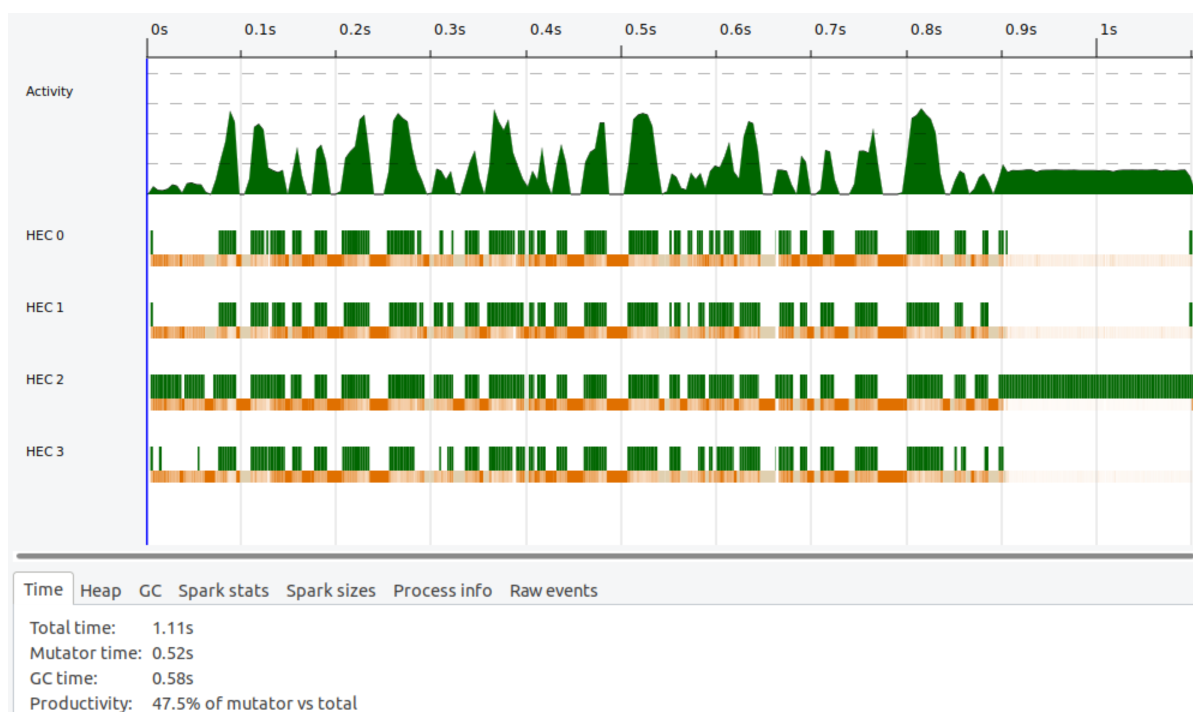

Figure 6: Threadscope of Parallel Iterative Fast Fourier Transform (length of polynomials: 20000).

3.4 Further Analysis

Figure 7 shows the runtime of three parallel algorithms vs. cores. As expected, runtime decreases as cores increases.

Figure 8 and Table 5 shows the runtime comparison of all implementations versus the length of polynomials. We can see that though parallelization helps, time complexity dominates. When length of polynomials is small (smaller than 3000), the parallel brute force algorithm can be faster than the sequential fast fourier transform. However, as length of polynomials increases, FFT wins and the difference of runtime between FFT and Brute Force gets larger and larger.

Also, from Figure 8 we can see that the runtime of all FFT implementation is stepwise. This is because that FFT needs to extend the length of polynomials to the closest power of 2, so polynomials with length

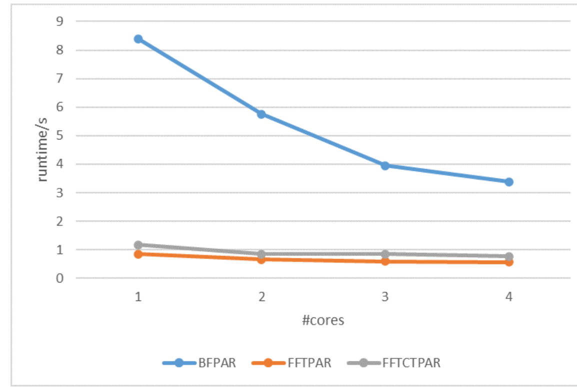

Figure 7: Runtime vs. cores (length of polynomials: 10000)

close to the same power of 2 will have similar runtime.

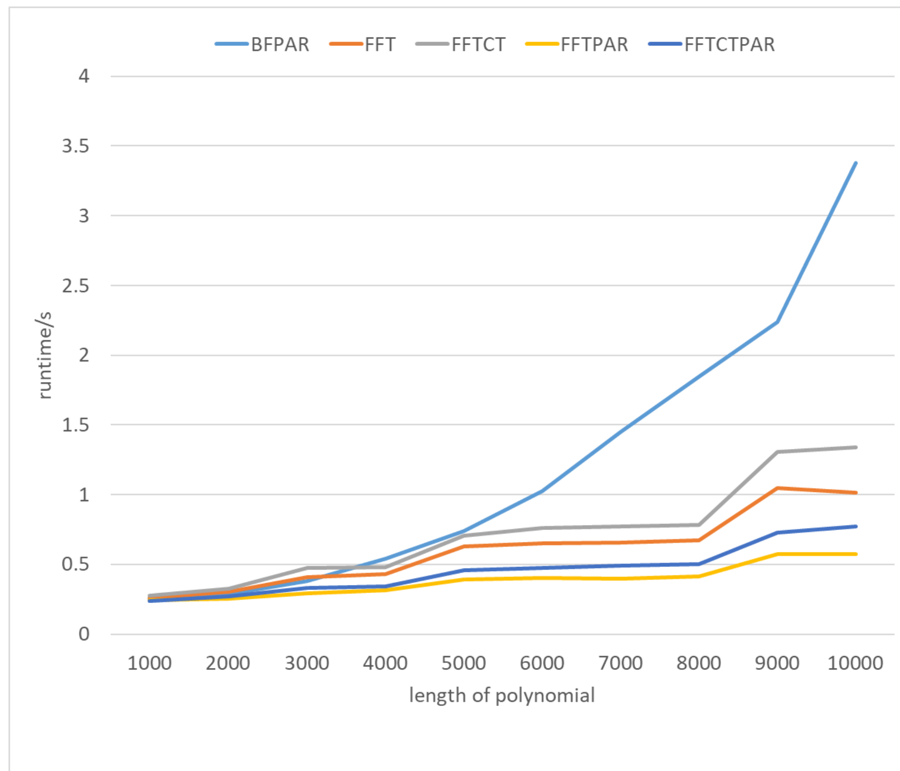

Figure 8: Runtime vs. length of polynomials.

Is it impossible for Brute Force implementation to beat FFT when length of polynomials is large? No. See Table 6. The above runtime analysis are all performed on polynomials with equal length. When the length of two polynomials for multiplication differ a lot, the sequential Brute Force implementation beats the parallel FFT implementation. This is because that FFT needs to extend the shorter polynomial to the same size as the longer one so that they can be mapped to the same frequency space. In this case, instead of $O(n^2)$ vs. $O(n \log n)$, the time complexity becomes $O(mn)$ vs. $O(n \log n)$, where m, n is the length of two polynomials and $m < n$.

length	BF	BFPAR	BFMR	FFT	FFTPAR	FFTCT	FFTCTPAR
1000	0.284	0.252	1.087	0.259	0.242	0.277	0.24
2000	0.605	0.287	3.512	0.303	0.255	0.324	0.273
3000	1.15	0.383	8.357	0.406	0.293	0.473	0.329
4000	2.151	0.542	killed	0.43	0.316	0.48	0.342
5000	2.89	0.74		0.629	0.39	0.705	0.458
6000	4.56	1.026		0.65	0.403	0.763	0.474
7000	7.41	1.45		0.656	0.396	0.772	0.493
8000	11.32	1.848		0.675	0.415	0.784	0.502
9000	15.52	2.239		1.047	0.575	1.305	0.727
10000	19.82	3.378		1.016	0.571	1.339	0.773
20000	89.564	16.693		2.1	1.048	2.644	1.412
30000	226.616	30.896		2.265	1.161	2.789	1.613
40000	423.338	85.157		4.079	1.932	5.288	2.724
50000	1341.603	151.333		4.296	2.102	5.416	2.866
60000	>1341.603	265.218		4.528	2.185	5.641	3.107
70000	>1341.604	411.472		8.469	3.7729	10.585	5.342
80000	>1341.605	643.82		8.536	3.928	11.028	5.421

Table 5: Runtime vs. length of polynomials.

length	BF	BFPAR	FFT	FFTPAR	FFTCT	FFTCTPAR
100,10000	0.361	0.341	0.987	0.559	1.189	0.825
100,100000	1.841	1.239	7.769	4.622	10.274	8.613

Table 6: Runtime with polynomials of unequal length.

4 References

1. Polynomial Multiplication
   `https://cse.hkust.edu.hk/mjg_lib/Classes/COMP3711H_Fall14/lectures/DandC_Multiplication_Handout.pdf`

2. Polynomial Multiplication via Fast Fourier Transforms
   `http://www.cs.toronto.edu/~denisp/csc373/docs/tutorial3-adv-writeup.pdf`

3. Parallel Fast Fourier Transform
   `https://courses.engr.illinois.edu/cs554/fa2015/notes/13_fft_8up.pdf`

4. Ocaml Implementation of Cooley-Tukey Algorithm
   `https://github.com/akabe/ocaml-numerical-analysis/blob/master/fft/fft.ml`

5. Computation graph of Cooley-Tukey Algorithm
   `http://www.akademik.ube.ege.edu.tr/~erciyes/CENG560/kumar/chap13.pdf`

6. Binary Exchange Algorithm
   `http://users.atw.hu/parallelcomp/ch13lev1sec2.html`

7. Fork/Join vs. Map Reduce `http://www.macs.hw.ac.uk/cs/techreps/docs/files/HW-MACS-TR-0096.pdf`

5 Code

app/Main.hs

```
module Main where

import System.IO(Handle, hGetLine, stdin, getLine, readLn, openFile, IOMode(ReadMode))

import qualified BruteForce.MultPoly(mult_polys)
import qualified BruteForce.MapReduce(mult_polys)
import qualified BruteForce.ParMultPoly(mult_polys)
import qualified FFT.FMultPoly(mult_polys)
import qualified FFT.CTMultPoly(mult_polys)
import qualified FFT.ParFMultPoly(mult_polys)
import qualified FFT.ParCTMultPoly(mult_polys)

import Data.Complex

get_nums :: Handle -> IO [Double]
get_nums = to_nums . hGetLine
  where
    to_nums line = do
      line_str <- line
      return $ map read $ words line_str

get_int :: IO Int
get_int = readLn

mult_polys :: String -> Int -> [Double] -> [Double] -> [Double]
mult_polys "BF" _ x y = BruteForce.MultPoly.mult_polys x y  -- sequential
mult_polys "BFMP" d x y = BruteForce.MapReduce.mult_polys d x y  -- parallel
mult_polys "BFPAR" d x y = BruteForce.ParMultPoly.mult_polys d x y -- parallel
mult_polys "FFT" _ x y = FFT.FMultPoly.mult_polys x y  -- sequential
mult_polys "FFTCT" _ x y = FFT.CTMultPoly.mult_polys x y  -- sequential
mult_polys "FFTPAR" d x y = FFT.ParFMultPoly.mult_polys d x y  -- parallel
mult_polys "FFTCTPAR" d x y = FFT.ParCTMultPoly.mult_polys d x y  -- parallel
mult_polys ver _ _ _ = error ("unknown version " ++ ver)

main :: IO ()
main = do
  _ <- putStrLn "version:"
  ver <- getLine
  _ <- putStrLn "input file path:"
  file_path <- getLine
  _ <- putStrLn "depth:"
  depth <- get_int
  handle <- System.IO.openFile file_path System.IO.ReadMode
  x <- get_nums handle
  y <- get_nums handle
  mapM_ print (mult_polys ver depth x y)
```

BruteForce/MultPoly.hs

```
-- Brute Force Polynomial Multiplication in Serialization

module BruteForce.MultPoly
```

```
(
  mult_poly_num
, add_polys
, mult_polys
) where

mult_poly_num :: [Double] -> Double -> [Double]
mult_poly_num [] _ = []
mult_poly_num (p:ps) num = (p * num) : (mult_poly_num ps num)

add_polys :: [Double] -> [Double] -> [Double]
add_polys x [] = x
add_polys [] y = y
add_polys (x:xs) (y:ys) = (x + y) : (add_polys xs ys)

mult_polys :: [Double] -> [Double] -> [Double]
mult_polys _ [] = []
mult_polys x (y:ys) = add_polys (mult_poly_num x y) (0 : mult_polys x ys)
```

BruteForce/MapReduce.hs

```
-- Brute Force Polynomial Multiplication in Parallelization with MapReduce

module BruteForce.MapReduce
(
  map_reduce
, mult_polys
) where

import qualified Data.Map as M
import Control.Parallel(pseq)
import Control.Parallel.Strategies(NFData, parMap, rdeepseq, parListChunk, runEval)

{-
map_reduce :: (NFData a, NFData b, NFData c, NFData d)
              => Int -> (a -> b) -> ([b] -> [c]) -> (c -> d) -> [a] -> [d]
map_reduce depth mapper shuffle reducer input =
  let par_size = (length input) `div` depth in
  let mapper_result = runEval £ parListChunk par_size rdeepseq (map mapper input) in
  let shuffle_result = shuffle mapper_result in
  let reducer_result = runEval £ parListChunk par_size rdeepseq (map reducer
↪  shuffle_result) in
  reducer_result
-}
map_reduce :: (NFData a, NFData b, NFData c, NFData d)
              => Int -> (a -> b) -> ([b] -> [c]) -> (c -> d) -> [a] -> [d]
map_reduce _ mapper shuffle reducer input =
  let mapper_result = parMap rdeepseq mapper input in
  let shuffle_result = shuffle mapper_result in
  let reduce_result = parMap rdeepseq reducer shuffle_result in
  mapper_result `pseq` reduce_result

preprocess_polys :: [Double] -> [Double] -> [(Int, Int, Double, Double)]
preprocess_polys l1 l2 = [(i1, i2, f1, f2) | (i1, f1) <- i_l1, (i2, f2) <- i_l2]
```

```
  where
    i_l1 = zip [1..(length l1)] l1
    i_l2 = zip [1..(length l2)] l2

mult_poly_mapper :: (Int, Int, Double, Double) -> (Int, [Double])
mult_poly_mapper (i1, i2, f1, f2) = (i1 + i2, [f1 * f2])

-- O(nlogn) overhead
mult_poly_shuffle :: [(Int, [Double])] -> [(Int, [Double])]
mult_poly_shuffle l = M.toList $ M.fromListWith (++) l

-- since shuffle_result comes from Map, it is already sorted
-- and therefore index can be discarded
mult_poly_reducer :: (Int, [Double]) -> Double
mult_poly_reducer (_, l) = sum l

mult_polys :: Int -> [Double] -> [Double] -> [Double]
mult_polys depth x y =
    map_reduce depth mult_poly_mapper mult_poly_shuffle mult_poly_reducer
  $ preprocess_polys x y
```

BruteForce/ParMultPoly.hs

```
-- Brute Force Polynomial Multiplication in Parallelization with Par
module BruteForce.ParMultPoly
(
  mult_polys
) where

import BruteForce.MultPoly(mult_poly_num, add_polys)
import Control.Parallel.Strategies(rseq, rpar, runEval)
import Control.DeepSeq(force)

mult_polys :: Int -> [Double] -> [Double] -> [Double]
mult_polys 0 x y = foldr shift_add [0] $ map (mult_poly_num x) y
  where
    shift_add a z = add_polys a (0 : z)
mult_polys d x y = runEval $ do
  a_prods <- rpar (force (mult_polys (d - 1) x ay))
  b_prods <- rpar (force (mult_polys (d - 1) x by))
  _ <- rseq a_prods
  _ <- rseq b_prods
  return $ add_polys a_prods ((take half_l $ repeat 0) ++ b_prods)
  where
    half_l = (length y) `div` 2
    (ay, by) = splitAt half_l y
```

FFT/FMultPoly.hs

```
-- Polynomial Multiplication using Recursive Fast Fourier Transform in Serialization

module FFT.FMultPoly
```

```haskell
(
  mult_polys,
  fft,
  ifft,
  convert,
  split,
  combine
) where

import Data.Complex

split :: [a] -> ([a], [a])
split l = split_helper l True
  where
    split_helper [] _ = ([], [])
    split_helper (x:xs) is_odd
      | is_odd = (x:o, e)
      | otherwise = (o, x:e)
      where (o, e) = split_helper xs (not is_odd)

combine :: [Complex Double] -> [Complex Double] -> Complex Double -> [Complex Double]
combine p1 p2 w = pf ++ ps
  where
    zip_p = zip3 [0..((length p1) - 1)] p1 p2
    pf = map (\(i, a, b) -> a + (w ** (fromIntegral i)) * b) zip_p
    ps = map (\(i, a, b) -> a - (w ** (fromIntegral i)) * b) zip_p

-- n is length of l and it must be power of 2;
-- w is nth root of unity: exp (2*pi*(0 :+ (-1))/n)
-- fft [1, 2, 3, 4, 5, 6, 7, 8] 8 (exp (-2*pi*(0:+1)/8))
fft :: [Complex Double] -> Int -> Complex Double -> [Complex Double]
fft l 1 _ = l
fft l n w = combine p1 p2 w
  where
    (l1, l2) = split l
    p1 = fft l1 (n `div` 2) (w ** 2)
    p2 = fft l2 (n `div` 2) (w ** 2)

ifft :: [Complex Double] -> Int -> Complex Double -> [Complex Double]
ifft l n w = map (\x -> x / (fromIntegral n)) (fft l n (1 / w))

convert :: [Double] -> Int -> [Complex Double]
convert x l = map (\f -> (f :+ 0)) (x ++ (take l (repeat 0)))

mult_polys :: [Double] -> [Double] -> [Double]
mult_polys x y =
  take (length_x + length_y - 1) $ map (\a -> realPart a) (ifft fft_r n w)
  where
    length_x = length x
    length_y = length y
    -- n > 2*l, n is power of 2
    n = 2 ^ (ceiling $ logBase 2 (fromIntegral (2 * (max length_x length_y))))
    w = exp (- 2 * pi * (0 :+ 1) / (fromIntegral n))
    fft_x = fft (convert x (n - length_x)) n w
```

```haskell
    fft_y = fft (convert y (n - length_y)) n w
    fft_r = map (\(a, b) -> a * b) (zip fft_x fft_y)
```

FFT/ParFMultPoly.hs

```haskell
-- Polynomial Multiplication using FFT in Parallelization with Par

module FFT.ParFMultPoly
(
  fft,
  mult_polys
) where

import Data.Complex
import Control.Parallel.Strategies(rseq, rpar, runEval)
import Control.DeepSeq(force, NFData)

import FFT.FMultPoly(split, convert)
import qualified FFT.FMultPoly(fft)

-- without paring combine, at the end there is a long period that only one core works
-- unable to do the same thing on split
combine :: [Complex Double] -> [Complex Double] -> Complex Double -> [Complex Double]
combine p1 p2 w = runEval $ do
  pf <- rpar (force (map (\(i, a, b) -> a + (w ** (fromIntegral i)) * b) zip_p))
  ps <- rpar (force (map (\(i, a, b) -> a - (w ** (fromIntegral i)) * b) zip_p))
  _ <- rseq pf
  _ <- rseq ps
  return $ pf ++ ps
  where
    zip_p = zip3 [0..((length p1) - 1)] p1 p2

-- parallel fft with depth
fft :: [Complex Double] -> Int -> Complex Double -> Int -> [Complex Double]
fft l 1 _ _ = l
fft l n w 0 = FFT.FMultPoly.fft l n w
fft l n w d = runEval $ do
  p1 <- rpar (force (fft l1 (n `div` 2) (w ** 2) (d - 1)))
  p2 <- rpar (force (fft l2 (n `div` 2) (w ** 2) (d - 1)))
  _ <- rseq p1
  _ <- rseq p2
  return $ combine p1 p2 w
  where
    (l1, l2) = split l

ifft :: [Complex Double] -> Int -> Complex Double -> Int -> [Complex Double]
ifft l n w d = map (\x -> x / (fromIntegral n)) (fft l n (1 / w) d)

mult_polys :: Int -> [Double] -> [Double] -> [Double]
mult_polys depth x y =
  take (length_x + length_y - 1) $ map (\a -> realPart a) (ifft fft_r n w depth)
  where
    length_x = length x
    length_y = length y
```

```
    -- n > 2*l, n is power of 2
    n = 2 ^ (ceiling $ logBase 2 (fromIntegral (2 * (max length_x length_y))))
    w = exp (- 2 * pi * (0 :+ 1) / (fromIntegral n))
    fft_x = fft (convert x (n - length_x)) n w depth
    fft_y = fft (convert y (n - length_y)) n w depth
    fft_r = map (\(a, b) -> a * b) (zip fft_x fft_y)
```

FFT/CTMultPoly.hs

```
--  Polynomial Multiplication with Iterative Fast Fourier Transform (Cooley-Tukey
↪   Algorithm) in Serialization

module FFT.CTMultPoly
(
  mult_polys,
  iter_fft,
  inverse_iter_fft
) where

import Data.Complex
import FFT.FMultPoly(split, convert)

bit_reverse_l :: [a] -> [a]
bit_reverse_l [] = []
bit_reverse_l [x] = [x]
bit_reverse_l x =
  l_rev ++ r_rev
  where
    (left, right) = split x
    l_rev = bit_reverse_l left
    r_rev = bit_reverse_l right

split_l :: [a] -> Int -> ([a], [a])
split_l l interval = split_l_helper l interval 0
  where
    half_interval = interval `div` 2
    split_l_helper [] _ _ = ([], [])
    split_l_helper (x:xs) interval index
      | index `mod` interval < half_interval = (x:left, right)
      | otherwise = (left, x:right)
      where
        (left, right) = split_l_helper xs interval (index + 1)

combine_l :: [a] -> [a] -> Int -> [a]
combine_l left right interval = combine_l_helper left right interval 0
  where
    half_interval = interval `div` 2
    combine_l_helper [] [] _ _ = []
    combine_l_helper [] right _ _ = right
    combine_l_helper left [] _ _ = left
    combine_l_helper left@(l:ls) right@(r:rs) interval index
      | index `mod` interval < half_interval = l : (combine_l_helper ls right interval
      ↪  (index + 1))
      | otherwise = r : (combine_l_helper left rs interval (index + 1))
```

```haskell
iter_fft :: [Complex Double] -> [Complex Double]
iter_fft l =
  foldl fold_butterfly rev_l [1..num_bits]
  where
    len = length l
    num_bits = ceiling $ logBase 2 (fromIntegral len)
    rev_l = bit_reverse_l l  -- O(nlogn)

    -- O(n)
    butterfly m n l =
      combine_l fft_j_l fft_k_l n
      where
        half_n = n `div` 2
        half_len = (length l) `div` 2
        (j_l, k_l) = split_l l n
        w_l = map (\i -> (exp (-2 * pi * (0:+1) * (fromIntegral ((i `mod` half_n) * m)) /
        ↪  (fromIntegral len)))) [0..(half_len - 1)]
        j_k_w_l = zip3 j_l k_l w_l
        fft_j_l = map (\(j, k, w) -> j + w * k) j_k_w_l
        fft_k_l = map (\(j, k, w) -> j - w * k) j_k_w_l

    fold_butterfly l iter = butterfly (2 ^ (num_bits - iter)) (2 ^ iter) l

inverse_iter_fft :: [Complex Double] -> [Complex Double]
inverse_iter_fft l =
  map (\x -> (conjugate x) / (fromIntegral len)) fft_con_l
  where
    len = length l
    con_l = map conjugate l
    fft_con_l = iter_fft con_l

mult_polys :: [Double] -> [Double] -> [Double]
mult_polys x y =
  take (length_x + length_y - 1) $ map (\a -> realPart a) (inverse_iter_fft fft_r)
  where
    length_x = length x
    length_y = length y
    n = 2 ^ (ceiling $ logBase 2 (fromIntegral (2 * (max length_x length_y))))
    fft_x = iter_fft (convert x (n - length_x))
    fft_y = iter_fft (convert y (n - length_y))
    fft_r = map (\(a, b) -> a * b) (zip fft_x fft_y)
```

FFT/ParCTMultPoly.hs

```haskell
module FFT.ParCTMultPoly
(
  be_fft,
  splitChunks,
  mult_polys
) where

import Data.Complex
import Control.Parallel(pseq)
```

```haskell
import Control.Parallel.Strategies(rseq, rpar, runEval, parListChunk, parMap, rdeepseq,
↪  NFData, Eval)
import Control.DeepSeq(force)

import FFT.FMultPoly(split, convert)

bit_reverse_l :: NFData a => [a] -> Int -> [a]
bit_reverse_l [] _ = []
bit_reverse_l [x] _ = [x]
bit_reverse_l x 0 = (bit_reverse_l left 0) ++ (bit_reverse_l right 0)
  where (left, right) = split x
bit_reverse_l x d = runEval $ do
  l_rev <- rpar (force (bit_reverse_l left (d - 1)))
  r_rev <- rpar (force (bit_reverse_l right (d - 1)))
  _ <- rseq l_rev
  _ <- rseq r_rev
  return $ l_rev ++ r_rev
  where
    (left, right) = split x

split_l :: [a] -> Int -> ([a], [a])
split_l l interval = split_l_helper l interval 0
  where
    half_interval = interval `div` 2
    split_l_helper [] _ _ = ([], [])
    split_l_helper (x:xs) interval index
      | index `mod` interval < half_interval = (x:left, right)
      | otherwise = (left, x:right)
      where
        (left, right) = split_l_helper xs interval (index + 1)

combine_l :: [a] -> [a] -> Int -> [a]
combine_l left right interval = combine_l_helper left right interval half_interval 0
  where
    half_interval = interval `div` 2
    combine_l_helper [] [] _ _ _ = []
    combine_l_helper [] right _ _ _ = right
    combine_l_helper left [] _ _ _ = left
    combine_l_helper left@(l:ls) right@(r:rs) interval half_interval index
      | index < half_interval = l : (combine_l_helper ls right interval half_interval
      ↪  (index + 1))
      | index == interval - 1 = r : (combine_l_helper left rs interval half_interval 0)
      | otherwise = r : (combine_l_helper left rs interval half_interval (index + 1))

splitChunks :: Int -> [a] -> [[a]]
splitChunks _ [] = []
splitChunks n l = f : splitChunks n s
  where
    (f, s) = splitAt n l

butterfly :: Int -> Int -> [Complex Double] -> Int -> [Complex Double]
butterfly m n l len = combine_l fft_j_l fft_k_l n
  where
    half_n = n `div` 2
```

```haskell
    half_len = (length l) `div` 2
    (j_l, k_l) = split_l l n
    w_l = map (\i -> (exp (-2 * pi * (0:+1) * (fromIntegral ((i `mod` half_n) * m)) /
    ↪  (fromIntegral len)))) [0..(half_len - 1)]
    j_k_w_l = zip3 j_l k_l w_l
    fft_j_l = map (\(j, k, w) -> j + w * k) j_k_w_l
    fft_k_l = map (\(j, k, w) -> j - w * k) j_k_w_l

par_butterfly :: Int -> Int -> [[Complex Double]] -> Int -> Int -> [[Complex Double]]
par_butterfly iter_start iter_end par_lists num_bits len =
  parMap rdeepseq par_butterfly_func par_lists
  where
    fold_butterfly l iter = butterfly (2 ^ (num_bits - iter)) (2 ^ iter) l len
    par_butterfly_func l = foldl fold_butterfly l [iter_start..iter_end]

butterfly_interact :: Int -> Int -> [Complex Double] -> Int -> Int -> [Complex Double]
butterfly_interact m n l par_size len = runEval $ do
  fft_j_l <- parListChunk par_size rdeepseq (map (\(j, k, w) -> j + w * k) j_k_w_l)  --
  ↪  not balanced in the end
  fft_k_l <- parListChunk par_size rdeepseq (map (\(j, k, w) -> j - w * k) j_k_w_l)
  _ <- rseq fft_j_l
  _ <- rseq fft_k_l
  return $ combine_l fft_j_l fft_k_l n
  where
    half_n = n `div` 2
    half_len = (length l) `div` 2
    (j_l, k_l) = split_l l n
    w_l = map (\i -> (exp (-2 * pi * (0:+1) * (fromIntegral ((i `mod` half_n) * m)) /
    ↪  (fromIntegral len)))) [0..(half_len - 1)]
    j_k_w_l = zip3 j_l k_l w_l

be_fft  :: [Complex Double] -> Int -> [Complex Double]
be_fft l num_c = end_list `pseq` interact_list  -- TODO
  where
    len = length l
    num_bits = ceiling $ logBase 2 (fromIntegral len)
    -- take min to prevent num_c > length of list
    num_c_bits = ceiling $ logBase 2 (fromIntegral (min num_c len))
    c_partition_size = 2 ^ (num_bits - num_c_bits)

    fold_butterfly_interact l iter = butterfly_interact (2 ^ (num_bits - iter)) (2 ^
    ↪  iter) l (c_partition_size `div` 2) len

    rev_l = bit_reverse_l l num_c -- O(nlogn)

    split_lists = splitChunks c_partition_size rev_l
    end_list = concat (par_butterfly 1 (num_bits - num_c_bits) split_lists num_bits len)
    interact_list = foldl fold_butterfly_interact end_list [(num_bits - num_c_bits +
    ↪  1)..num_bits]

inverse_be_fft :: [Complex Double] -> Int -> [Complex Double]
inverse_be_fft l num_c =
  map (\x -> (conjugate x) / (fromIntegral len)) fft_con_l
  where
```

```
    len = length l
    con_l = map conjugate l
    fft_con_l = be_fft con_l num_c

mult_polys :: Int -> [Double] -> [Double] -> [Double]
mult_polys num_c x y =
  take (length_x + length_y - 1) $ map (\a -> realPart a) (inverse_be_fft fft_r num_c)
  where
    -- num_c = 4
    length_x = length x
    length_y = length y
    num_bits = ceiling $ logBase 2 (fromIntegral (2 * (max length_x length_y)))
    n = 2 ^ num_bits
    fft_x = be_fft (convert x (n - length_x)) num_c
    fft_y = be_fft (convert y (n - length_y)) num_c
    fft_r = map (\(a, b) -> a * b) (zip fft_x fft_y)
```