Project Report: The Discrete Logarithm Problem

Hana Mizuta (hm2694) and Michelle Mao (mm4957)

ALGORITHM

The discrete logarithm problem [$base^{exp} = x$ (mod m)] is a well-known problem in number theory: given *base*, *x*, and *m*, calculate *exp*. This problem is often used in cryptography — there is no efficient (or polynomial) solution to this problem, and with a large enough *m* it would take exponential time and a near-infinite amount of memory to try and crack the cipher.

We implemented the baby-step giant-step meet-in-the-middle algorithm to solve this problem. Here's an overview of the algorithm:

1. Calculate the ceiling of the square root of *m*; call this *sqrtM*
2. For each *i* from 1 to *sqrtM-1*, calculate $base^i$ mod *m* and store this value in a pair with *i* (key $base^i$ mod *m*, value *i*); call this *lhsTable*
3. For each *j* from 1 to *sqrtM*, calculate [(x)($base^{-j * sqrtM}$ mod m)] mod *m*; call this *rhsSol* and compare this value with each key in *lhsTable*
    a. If this value matches any key in *lhsTable*, return *lhsTable*[key] + (*j* * *sqrtM*); **this is the solution *exp***
4. If no values match for each pair *i, j*, there are no solutions to the problem with the given values for *base, x,* and *m*.

**The value $base^{-j * sqrtM}$ mod *m* is known as the *modular multiplicative inverse*, and satisfies the following property: [($base$ mod m)($base^{-1}$ mod m)] mod m = 1. The modular multiplicative inverse is only defined when *base* and *m* are relatively prime.

CODE

DLog.hs

The function *runBabyStepGiantStep* takes in a String of input and parses the input into *x, base,* and *m*, then passes these values into *babyStepGiantStep. babyStepGiantStep* starts the actual algorithm: first it checks if *base* and *m* are relatively prime, and returns a Left string if they are not. *sqrtM* is calculated, as well as *lhsTable* (using *powMod*, a function that returns b^e mod *m* when given *b, e,* and *m*). In this implementation, *lhsTable* is a list of tuples (key, value).

```haskell
runBabyStepGiantStep :: String -> Either String Integer
runBabyStepGiantStep line =
  let [x, base, modulus] = map read $ words line
  in babyStepGiantStep base x modulus

babyStepGiantStep :: Integer -> Integer -> Integer -> Either String Integer
babyStepGiantStep base x m
    | isRelativelyPrime base m == False = Left "Base and modulus must be relatively prime"
    | otherwise                         = babyStepGiantStep' base x m sqrtM 1 lhsTable
          where sqrtM                  = ceiling $ sqrt $ fromIntegral m
                lhsTable               = [(powMod base i m, i) | i <- [ 1 .. (sqrtM - 1) ]]
```

runBabyStepGiantStep and babyStepGiantStep

From here, *babyStepGiantStep* kicks off Step 3 of the algorithm by calling *babyStepGiantStep'*. Each time *babyStepGiantStep'* is called, *j* (called *numIter*) counts up by 1. *babyStepGiantStep'* uses the helper function *inverseEuclid*, which calculates the modular multiplicative inverse of an input *base* and *m*. With each *numIter*, first *lhsTable* is scanned to see if *rhsSol* is in the list of values; if it is, the solution *ans* is calculated and returned as a Right. The iterations stop when *numIter* passes *sqrtM*, otherwise the search continues.

```haskell
babyStepGiantStep' :: Integer -> Integer -> Integer -> Integer -> Integer -> [(Integer, Integer)] -> Either String Integer
babyStepGiantStep' base x m sqrtM numIter lhsTable
    | elem rhsSol (map fst lhsTable) = Right ans
    | numIter > sqrtM                = Left "No solution"
    | otherwise                       = babyStepGiantStep' base x m sqrtM (numIter + 1) lhsTable
            where rhsSol             = x * (inverseEuclid (base ^ (currRhsIdx)) m) `mod` m
                  ans                 = currLhsIdx + currRhsIdx
                  currLhsIdx        = getI rhsSol lhsTable
                  currRhsIdx        = sqrtM * numIter
```

babyStepGiantStep'

runDLog.hs

Finally, *runBabyStepGiantStep* is called in *runDLog.hs.* We were primarily concerned with the runtime of the algorithm on a large number of inputs, so the final output is the number of problems that have solutions in the file rather than the list of solutions itself.

```haskell
  let line      = lines file
      sol       = map (runBabyStepGiantStep) line
      -- sol       = map (runBabyStepGiantStepPar) line -- `using` parBuffer 1000 rseq -- TODO
      -- sol       = map (runBabyStepGiantStepParWithChunks) line -- `using` parListChunk 10000 rpar

  putStrLn $ show $ length $ filter isRight sol
```

runDLog.hs

CONSIDERATIONS

The runtime of this problem grows exponentially as the number of digits in *m* increase. We decided to test files with 7,500 lines of input with 4-digit *m*s, and 75 lines of input with 5-digit *m*s, because both had sequential runtimes of ~10s.

Run sequentially, the 4-digit *m* file (small.txt) took **12.151s** and the 5-digit *m* file (large.txt) took **12.789s**.

PARALLELIZING

The obvious route was to parallelize how the algorithm was being called. *`using` parList deepseq* had negligible effect on the elapsed times, so the values are omitted. (All times listed are the total elapsed times.)

	1 CORE	**2 CORE**	**4 CORE**	**8 CORE**
`using` parList rpar				
small.txt	11.954s	6.38s	3.56s	2.16s
spark number	15000	15000	15000	15000
sparks converted5000	15000	15000	15000	1
large.txt	12.877s	7.23s	5.02s	4.37s
spark number	150	150	150	150
sparks converted	150	150	150	150
`using` parList rseq				
small.txt	12.075s	6.18s	3.44s	2.17s
spark number	7500	7500	7500	7500
sparks converted	7500	7500	7500	7500
large.txt	13.403s	7.18s	4.84s	4.30s
spark number	75	75	75	75
sparks converted	75	75	75	75

Parallelizing how the algorithm is called with two different strategies

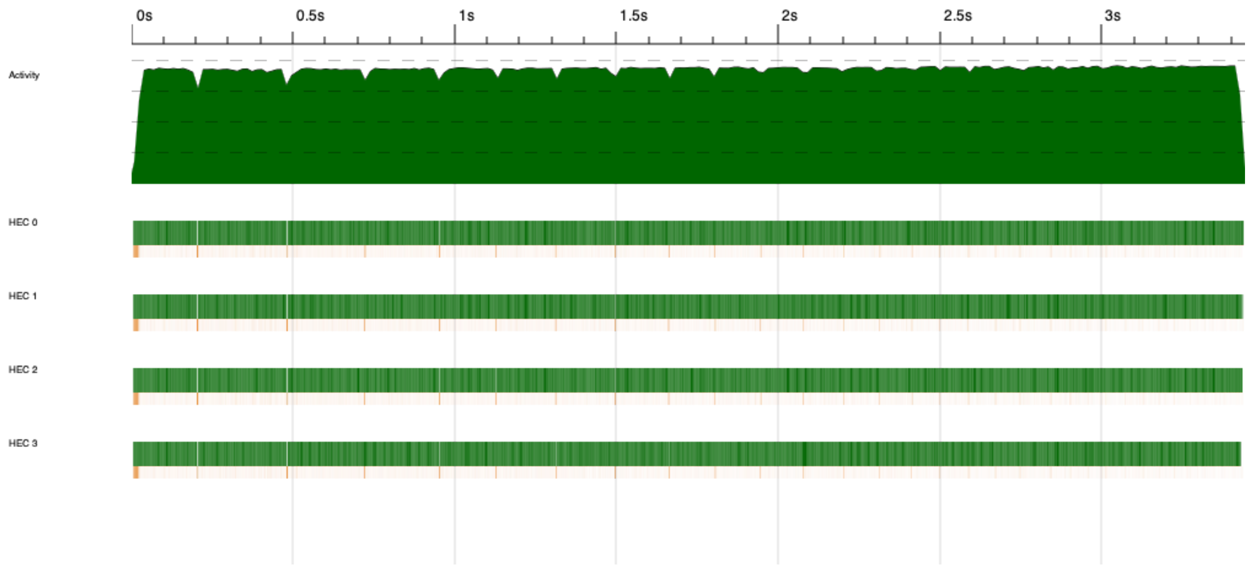
As we can see, this caused significant speedups. Running the parallelized version on 1 core took a bit longer than just running the sequential version, but didn't increase the elapsed time significantly. Running *`using` parList rpar* on small.txt gives a speedup of 12.151/6.38 = **1.90** on 2 cores, a speedup of 12.151/3.56 = **3.41** on 4 cores, and a speedup of 12.151/2.16 = **5.63** on 8.

	1 CORE	**2 CORE**	**4 CORE**	**8 CORE**
small.txt, rpar	1.02	1.90	3.41	5.63
large.txt, rpar	0.99	1.77	2.55	2.93
small.txt, rseq	1.01	1.97	3.53	5.60
large.txt, rseq	0.95	1.78	2.64	2.97

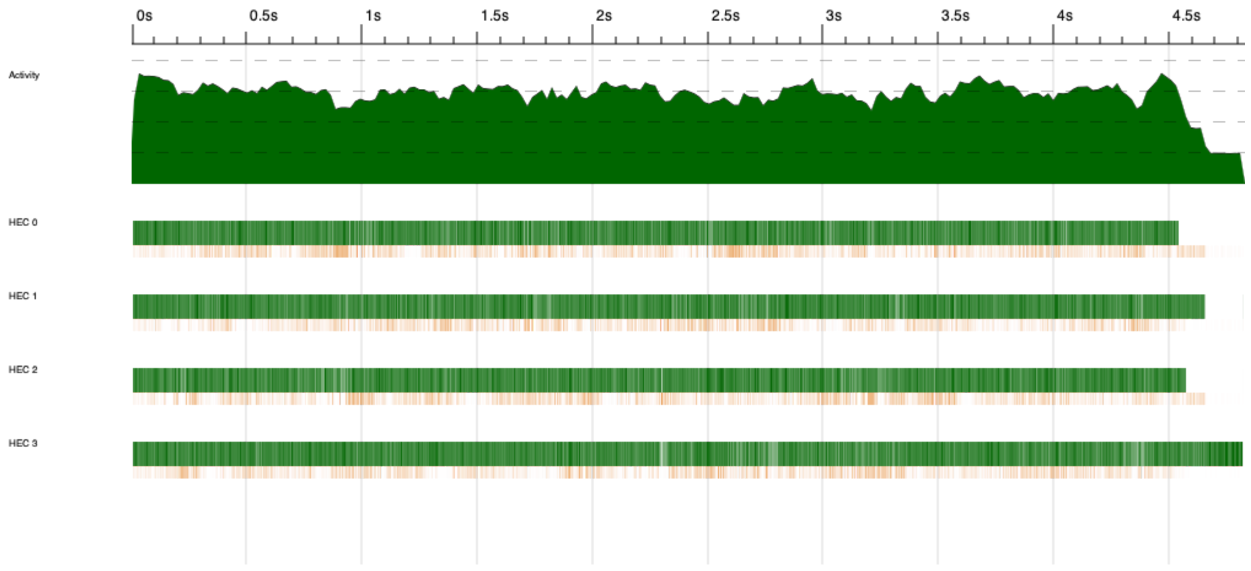
Speedups

The spark statistics look pretty good — a large number of sparks are being created in each case, particularly in small.txt (with 7500 input lines), but 100% of them are being converted. It's interesting to note that the *ParList rseq* strategy results in half the number of sparks (one for each line of input rather than two), but it doesn't seem to have a significant effect. This is most likely because rseq is performing the same evaluation as rpar, since each individual spark is its own problem (and not a list of values to map over).

Here's what Threadscope shows:



small.txt, parList rpar, 4 cores



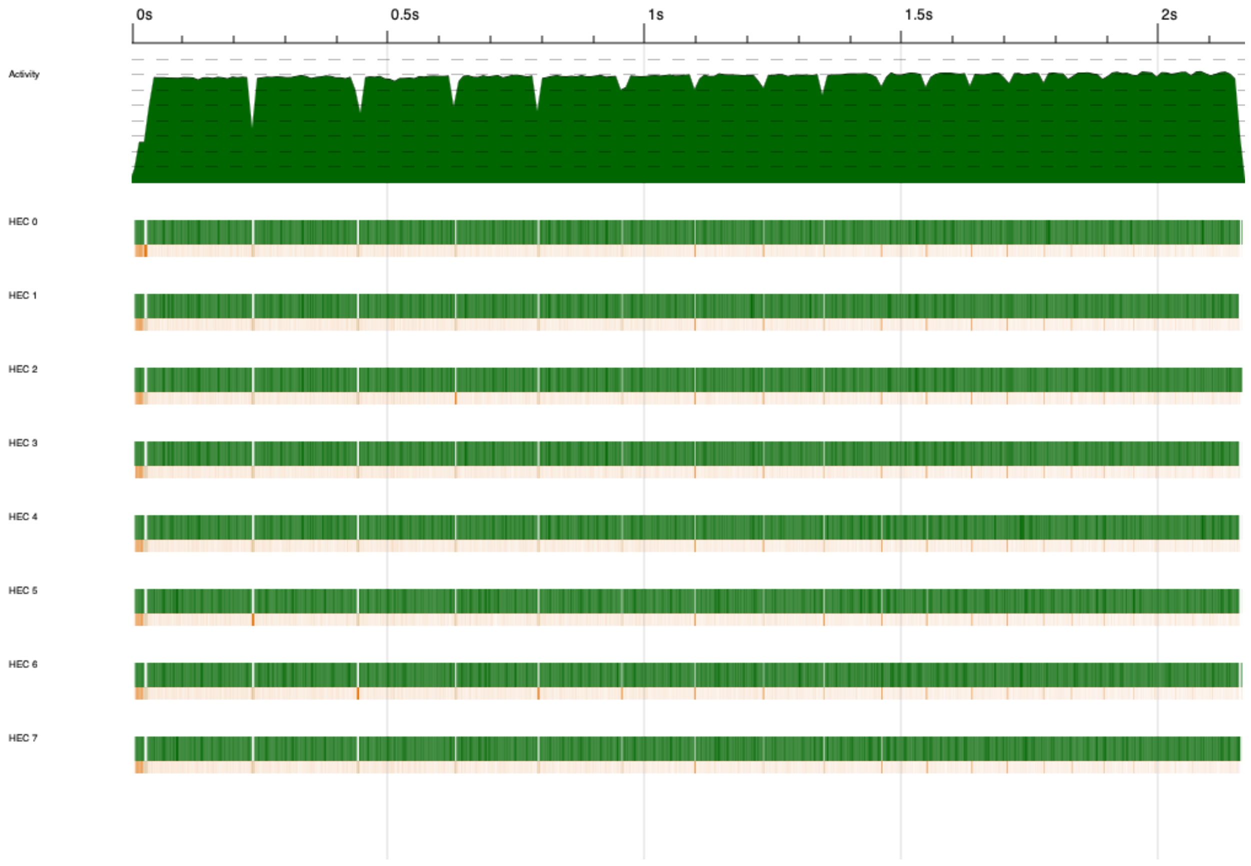
large.txt, parList rpar, 4 cores

The images look pretty good, as well — when we look at the activity for small.txt on 4 cores with *parList rpar*, the program seems to be utilizing the machine's resources well when dynamically partitioning the problem. For large.txt, there seems to be much more garbage collection breaking up the activity for all cores. This stays consistent regardless of the number of cores used (activity for 8 cores is below), which we expected: 4-digit moduli is handled well by the baby-step giant-step algorithm, but adding an extra digit increases the runtime for each individual problem significantly.

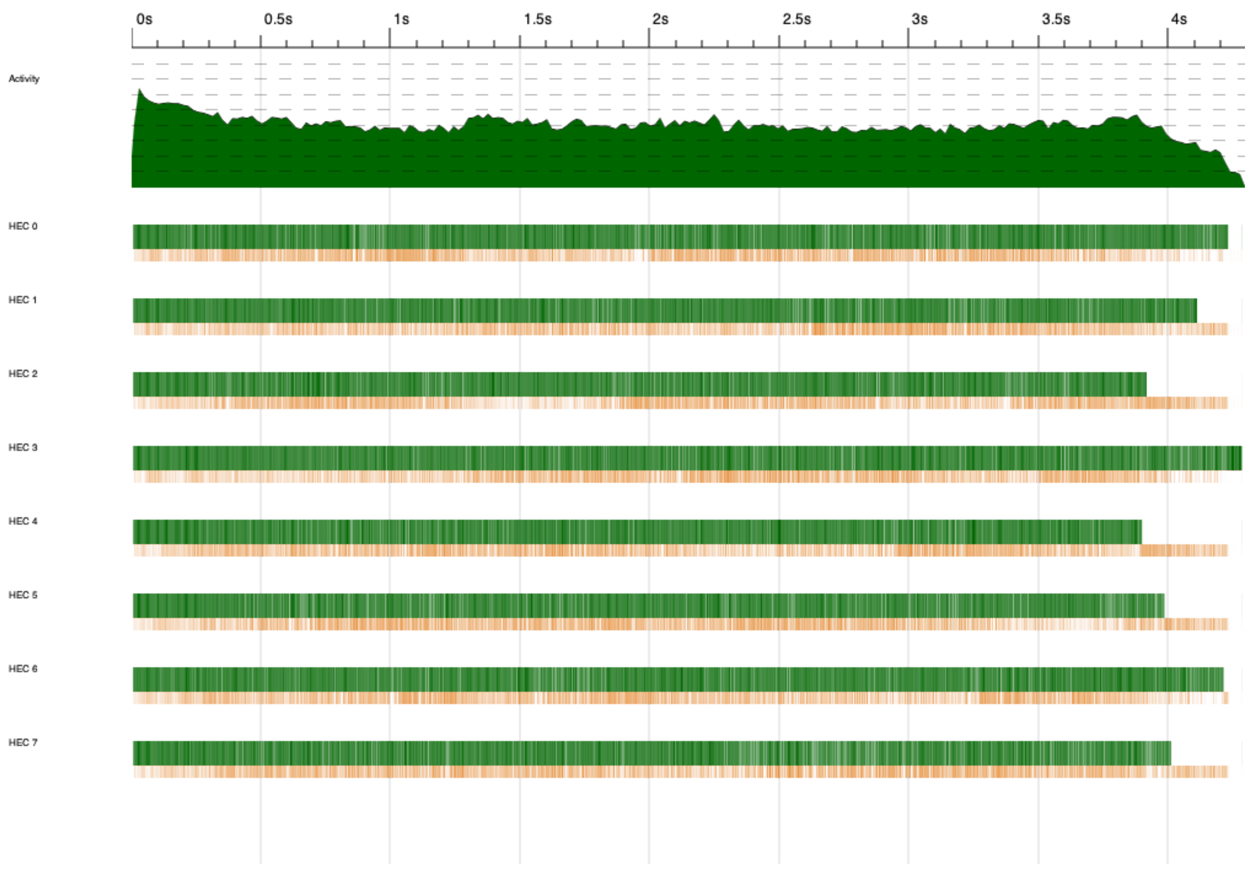
**Each problem has at minimum a runtime of O(*sqrtM*sqrtM*). For 4-digit *m* problems this can be anywhere from ~1000-9999, and for 5-digit *m* problems this balloons to ~10000-99999.

Because rpar and rseq (as well as deepseq) seem to evaluate the problems in pretty much the same manner, it doesn't seem like there's much else we can do to parallelize how the baby-step giant-step algorithm is called.

The next thing to try is parallelizing the algorithm itself!



small.txt, parList rseq, 8 cores



large.txt, rseq, 8 cores

PARALLELIZING THE ALGORITHM

When we inspect the algorithm, the largest time-sink is the searching of the computed value *rhsSol* (for *sqrtM* iterations) over our 'map' *lhsTable*, which is of size *sqrtM-1*. What seems to make the most sense is sparking off each iteration of *numIter* (*j* in Step 3) so that the search can be parallelized. In the parallel version, *babyStepGiantStepPar'* mostly stays the same: the only change is Left "no solution this iteration" is returned when there is no solution rather than Left "no solution".

```
babyStepGiantStepPar' :: Integer -> Integer -> Integer -> Integer -> [(Integer, Integer)] -> Integer -> Either String Integer
babyStepGiantStepPar' base x m sqrtM lhsTable numIter
  | elem rhsSol (map fst lhsTable) = Right ans
  | otherwise                      = Left "No match this iteration"
          where rhsSol             = x * (inverseEuclid (base ^ (currRhsIdx)) m) `mod` m
                ans                = currLhsIdx + currRhsIdx
                currLhsIdx         = getI rhsSol lhsTable
                currRhsIdx         = sqrtM * numIter
```

babyStepGiantStepPar'

The list *iterMap* holds the list [1..*numIter*], and *babyStepGiantStepPar'* is now mapped over *iterMap*. To check if a solution was found, *allRight* is called on the resulting list of Eithers and will contain the solution if a Right is present. If the result contains only Lefts, there was no solution.

```haskell
babyStepGiantStepPar :: Integer -> Integer -> Integer -> Either String Integer
babyStepGiantStepPar base x m
  | isRelativelyPrime base m == False        = Left "Base and modulus must be relatively prime"
  | length allRight > 0                   = head allRight
  | otherwise                             = Left "No solution"
      where sqrtM                         = ceiling $ sqrt $ fromIntegral m
            lhsTable                      = [(powMod base i m, i) | i <- [ 1 .. (sqrtM - 1) ]]
            iterMap                       = [1 .. sqrtM]
            allSols                       = map (babyStepGiantStepPar' base x m sqrtM lhsTable) iterMap `using` parList rpar
            allRight                      = filter isRight allSols
```
babyStepGiantStepPar

Here are the results of running *parList rpar* (and only the sequential *map (runBabyStepGiantStepPar) lines* in *runDLog.hs*):

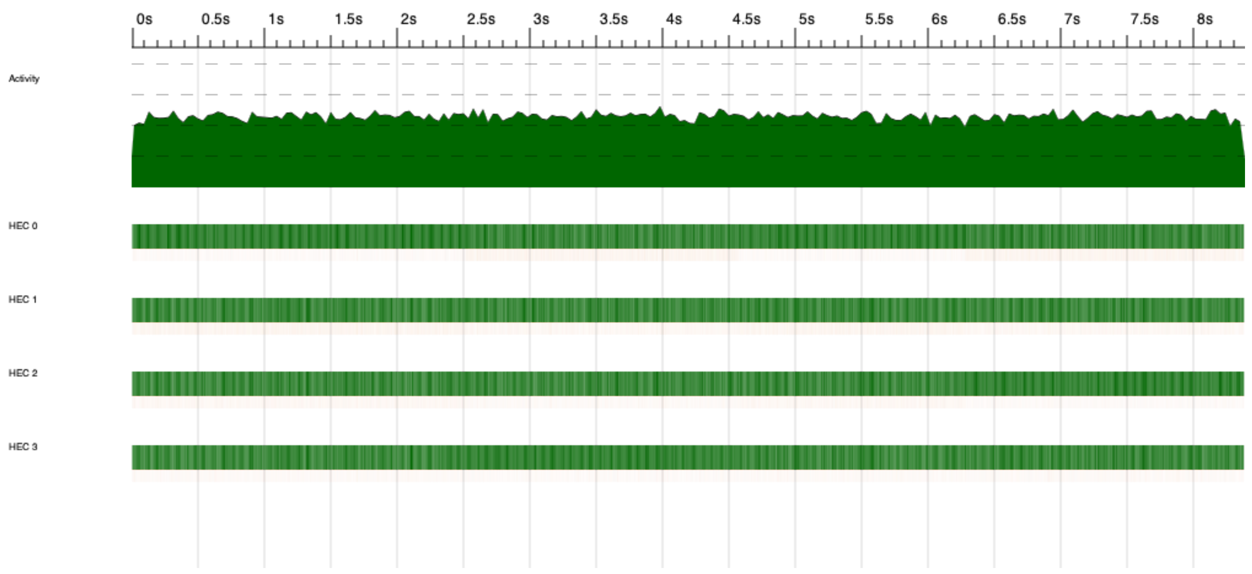
*Only *rpar* data is included because *rseq* performs nearly the same

	1 CORE	**4 CORE**
parList rpar		
small.txt	13.658s	8.37s
spark number		657496
sparks converted		290823
large.txt	14.142s	5.19s
spark number		22641
sparks converted		20227

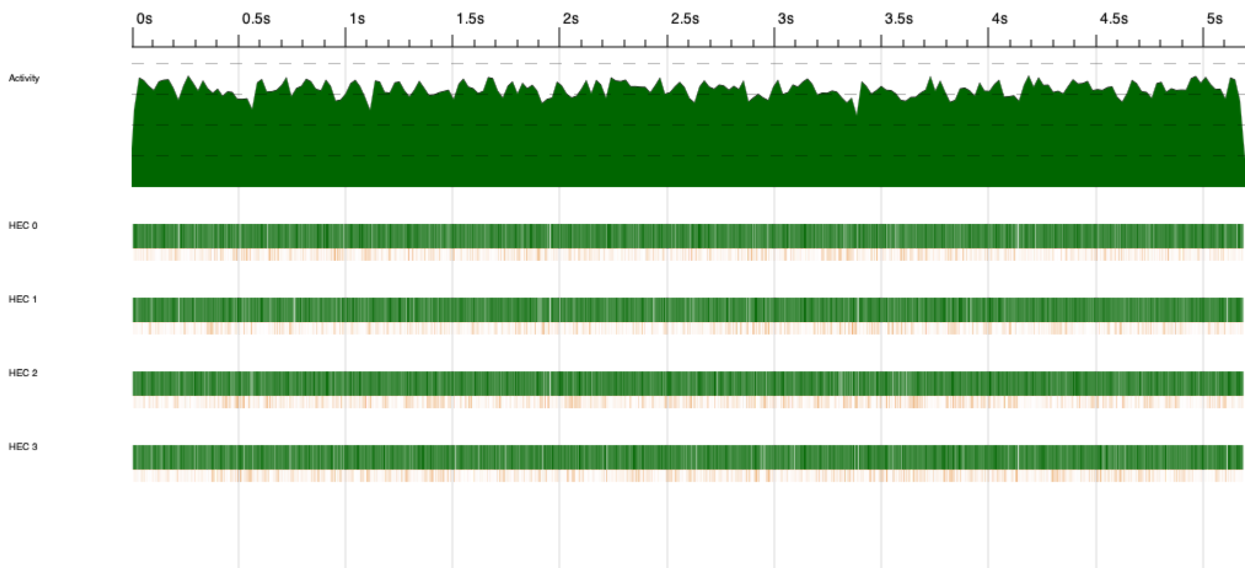
Parallelizing the algorithm (parList rpar)

The first thing to note is that running *babyStepGiantStepPar* sequentially already takes 1 to 2 more seconds than the normal sequential version, which makes sense because of the increased overhead. While 0 sparks overflowed for both, there were 44x the number of sparks for small, and 151x the number of sparks for large (!). For both small.txt and large.txt, many sparks were GC'd or had fizzled, where there previously had been 0. However, the speedups as well as the spark statistics show some interesting results: small.txt had a speedup of 12.151/8.37 = 1.45 and large.txt had a speedup of 12.789/5.19 = 2.46, while 44.2% of small.txt's sparks and 89% of large.txt's sparks converted.

Let's take a look at the activity of each:



small.txt, parList rpar, 4 cores (algorithm)



large.txt, parList rpar, 4 cores (algorithm)

In contrast with the previous parallelization, there is more machine activity on the 5-digit moduli file than on the 4-digit moduli file. This is what we expected — parallelizing the algorithm should benefit large more, because the bulk of the runtime is spent on solving the problems. With 8 cores, for small.txt it's 12.151/7.057 = 1.72 and for large.txt it's 12.789/4.000 = 3.20.

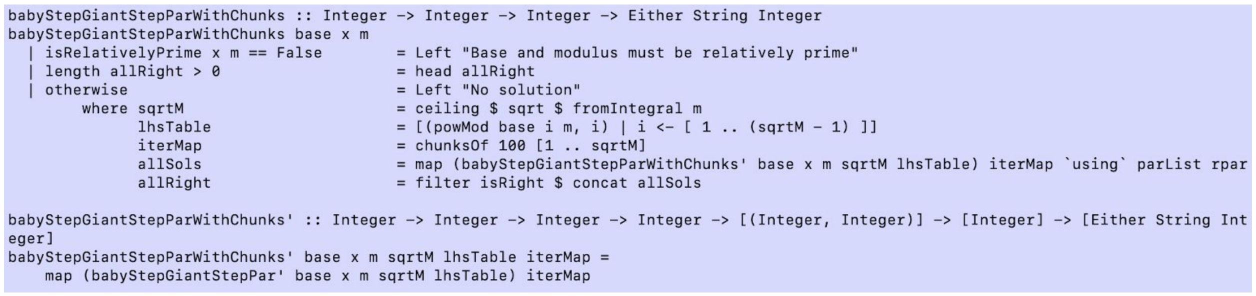
While these speedups are much smaller than parallelizing the calling of the problem, they still speed up the runtime of the program significantly (considering the speedups make up for the increased overhead of parallelizing the algorithm). Parallelizing the algorithm also speeds up 5-digit *m* problems more than 4-digit *m* problems, which is what we were aiming for.

**We also tried parBuffer with chunks of different sizes (100, 50, 20), but too many sparks were still being created and there was less speedup.

The next thing to try is parallelizing not every single iteration in *iterMap*, but chunks. This should reduce the spark pool to hopefully better numbers.

USING CHUNKS

This is accomplished in *babyStepGiantStepParWithChunks*: *iterMap* is now a list of lists of Integers, and *babyStepGiantStepParWithChunks'* maps *babyStepGiantStepPar'* over the inside lists of Integers. Finally, *allRight* combines the results using concat.



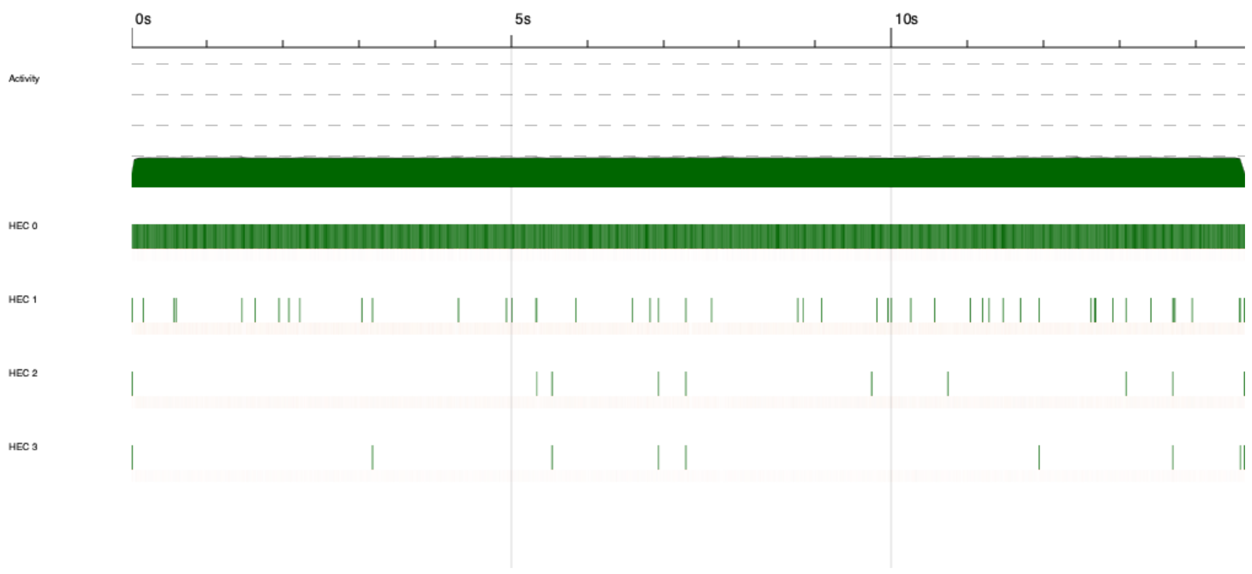
```
babyStepGiantStepParWithChunks :: Integer -> Integer -> Integer -> Either String Integer
babyStepGiantStepParWithChunks base x m
  | isRelativelyPrime x m == False        = Left "Base and modulus must be relatively prime"
  | length allRight > 0                   = head allRight
  | otherwise                             = Left "No solution"
      where sqrtM                         = ceiling $ sqrt $ fromIntegral m
            lhsTable                      = [(powMod base i m, i) | i <- [ 1 .. (sqrtM - 1) ]]
            iterMap                       = chunksOf 100 [1 .. sqrtM]
            allSols                       = map (babyStepGiantStepParWithChunks' base x m sqrtM lhsTable) iterMap `using` parList rpar
            allRight                      = filter isRight $ concat allSols

babyStepGiantStepParWithChunks' :: Integer -> Integer -> Integer -> Integer -> [(Integer, Integer)] -> [Integer] -> [Either String Int
eger]
babyStepGiantStepParWithChunks' base x m sqrtM lhsTable iterMap =
    map (babyStepGiantStepPar' base x m sqrtM lhsTable) iterMap
```

babyStepGiantStepParWithChunks and babyStepGiantStepParWithChunks'

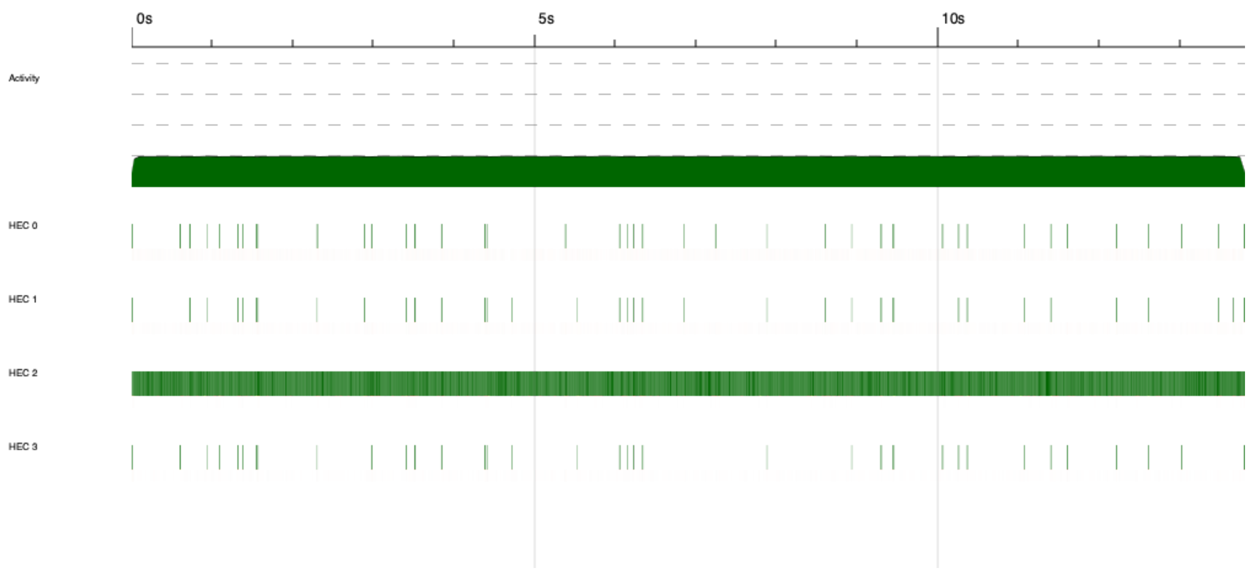
Here are the results:

	1 CORE	**4 CORE**
parList rpar		
small.txt	13.668s	14.46s
spark number		9158
sparks converted		13 (9030 GC'd)
large.txt	12.134s	12.37
spark number		230
sparks converted		146 (78 GC'd)

Parallelizing the algorithm (chunk size 100)



small.txt, babyStepGiantStepWithChunks, chunk size 100, 4 cores



large.txt, babyStepGiantStepWithChunks, chunk size 100, 4 cores

This is really bad! The number of sparks significantly decreased, but the number of converted sparks decreased even more. It seems that the additional overhead of breaking *iterMap* and later calling *concat* to put it back together likely resulted in the decreased speedup (compared to *babyStepGiantStepPar*).

Chunking *iterMap* didn't work at all, so all that's left is attempting to combine the two earlier strategies (parallelizing both how the algorithm is called and the algorithm itself with *babyStepGiantStepPar*).

MULTIPLE LAYERS OF PARALLELISM

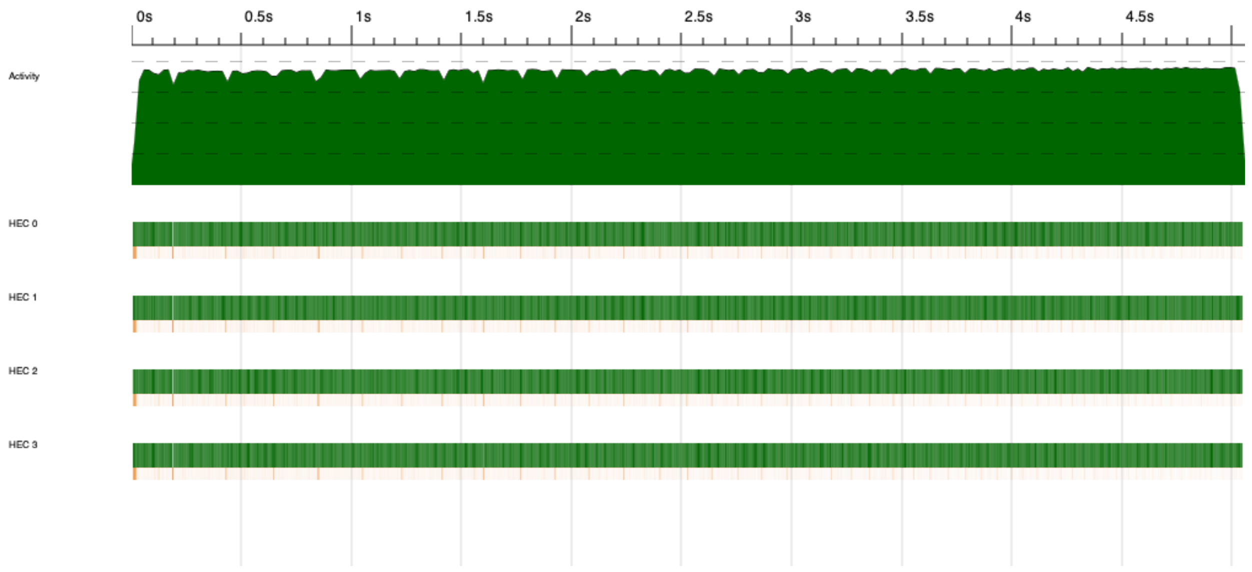
Here are the results with *parList rpar* in both *runDLog.hs* and *babyStepGiantStepPar*:

	1 CORE	**2 CORE**	**4 CORE**	**8 CORE**
double parList rpar				
small.txt	14.810s	8.36s	5.06s	3.12s
spark number		715980	681896	680404
sparks converted5000		15430	119515	42337
large.txt	14.147s	8.31s	5.35s	4.74s
spark number		22953	22788	22788
sparks converted		312	368	566

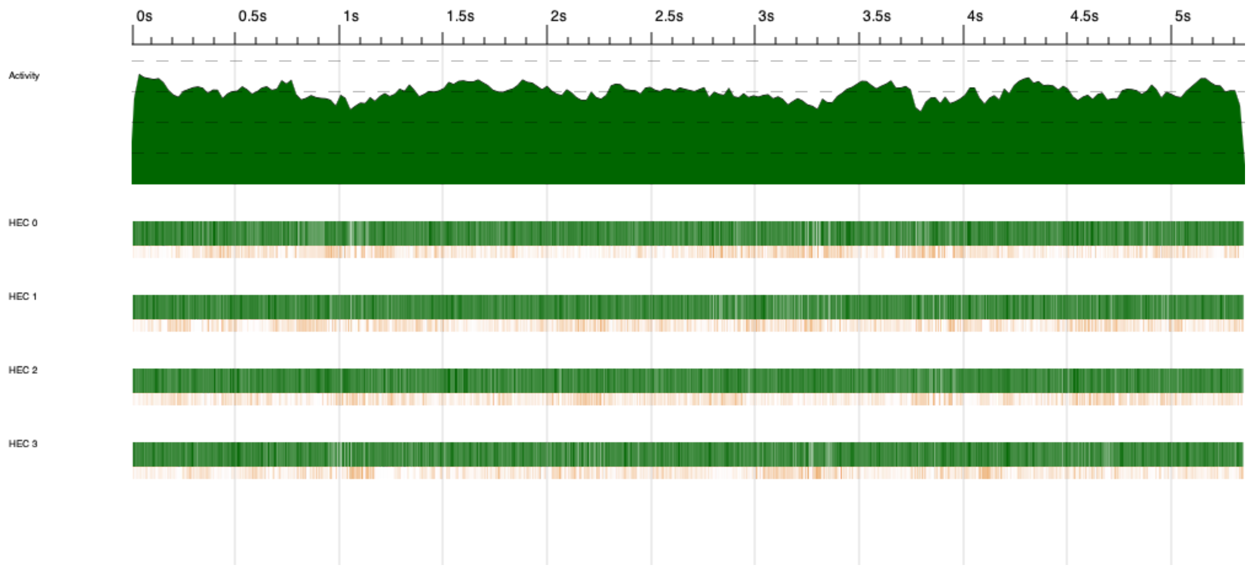
double parList rpar

	1 CORE	**2 CORE**	**4 CORE**	**8 CORE**
small.txt	0.82	1.45	2.40	3.89
large.txt	0.90	1.54	2.39	2.70

Speedups with double parList rpar



small.txt, double parList rpar, 4 cores



large.txt, double parList rpar, 4 cores

The results look decent — large.txt seems to maintain a similar speedup when compared with *babyStepGiantStepPar* (2.39 compared to 2.46), while small.txt is significantly sped up when compared to *babyStepGiantStepPar* (2.40 compared to 1.45). The activity is significantly higher for small.txt, which is good to see (because adding the *rpar* to the calling of *runBabyStepGiantStep* should speed up small.txt more).

CONCLUSION

We attempted to parallelize using different combinations of how the algorithm was being called (*runBabyStepGiantStep*) and the algorithm itself (*babyStepGiantStep*). First, we changed how *runBabyStepGiantStep* was called, and *`using` parList rpar* seemed to give the best results. Next, we tried parallelizing the algorithm in two ways: *babyStepGiantStepPar*, which created sparks for each iteration in *iterMap*, and *babyStepGiantStepParWithChunks*, which created sparks for chunks of iterations in *iterMap*. *babyStepGiantStepParWithChunks* failed miserably, so finally we combined changing how the algorithm was being called and using *babyStepGiantStepPar.*

With 4-digit moduli, changing only how *runBabyStepGiantStep* ran resulted in the best speedups (5.63 on 8 cores for 4-digit *m*s!). The speedup for 5-digits was around the same for changing how *runBabyStepGiantStepPar* ran, and using the parallel algorithm function *babyStepGiantStepPar* (with a speedup of 2.96 on 8 cores). 8 cores always had the largest speedup for our input files. Finally, the choice of which particular strategy or combination of strategies to use depends on how many digits are in the moduli of the input file.

Addendum: Regex

ALGORITHM

Initially, we wanted to parallelize regex matching. Oftentimes users may want to search through multiple files for a given word. Sometimes, however, users may want to match against a more general regex rather than just a simple given word. We wanted to find a way to regex match against multiple files in an efficient manner.

CODE

For regex matching, we first created 1000 files with 10000 words in each files (each file was on a new line) using Python. We then

1. Read through all of the 1000 files with readFile and saved the contents into a 1000 element list of Strings
2. Mapped matchRegex over the above list
3. Wrote the words that matched the regex into new files

We then tried to spark off step 2 using parMap. However, we found that there was a bottleneck with the IO. Step 1 took so long that Step 2 essentially became sequential.

To minimize the IO actions, we edited steps 1 and 3 to create the following steps:

1. Created a new file of 100 lines that had 10k words per line and saved the contents into a 100 element list of Strings.
2. Mapped matchRegex over the above list
3. Calculated the total number of regex word matches we found and printed to stdout

Again, we tried to spark off step 2 using parMap. However, we found that there was still a bottleneck with the IO, and Step 2 was still being executed sequentially.

We also tried to chunk (similar to how we did in the DLog with both chunksOf and with parChunks) the input, but that did not help.

README

```
=======================
DLOG
=======================
To run single threaded on macOS
$ ghc -O2 runDLog.hs -rtsopts -eventlog
$ ./runDLog input/small.txt +RTS -ls
$ ./threadscope.osx runDLog.eventlog

To run with 4 cores on macOS
$ ghc -O2 runDLog.hs -rtsopts -eventlog -threaded
$ ./runDLog input/small.txt +RTS -N4 -ls
$ ./threadscope.osx runDLog.eventlog

=======================
REGEX MATCHING
=======================
$ cd regex

To run single threaded on macOS
$ ghc -O2 runRegex.hs -rtsopts -eventlog
$ ./runRegex regex_in.txt at +RTS -s

To run with 4 cores on macOS
$ ghc -O2 runRegex.hs -rtsopts -eventlog -threaded
$ ./runRegex regex_in.txt at +RTS -N4 -s
```

runDLog.hs

```haskell
import Control.Parallel.Strategies
import Data.Either
import DLog
import System.Environment(getArgs)

main :: IO ()
main = do
  [f] <- getArgs
  file <- readFile f

  let line      = lines file
      sol       = map (runBabyStepGiantStep) line
      -- sol       = map (runBabyStepGiantStepPar) line `using` parList rseq -- TODO
      -- sol       = map (runBabyStepGiantStepParWithChunks) line `using` parList rseq -- TODO

  putStrLn $ show $ length $ filter isRight sol
```

DLog.hs

```
module DLog (runBabyStepGiantStep,
             runBabyStepGiantStepPar,
             runBabyStepGiantStepParWithChunks) where

import qualified Data.Bits as Bits (shift)
import Data.Either
import Data.List.Split
import Control.Parallel.Strategies

-- if parallel
runBabyStepGiantStepPar :: String -> Either String Integer
runBabyStepGiantStepPar line =
    let [x, base, modulus] = map read $ words line
    in babyStepGiantStepPar base x modulus

babyStepGiantStepPar :: Integer -> Integer -> Integer -> Either String Integer
babyStepGiantStepPar base x m
  | isRelativelyPrime base m == False     = Left "Base and modulus must be relatively prime"
  | length allRight > 0                   = head allRight
  | otherwise                             = Left "No solution"
        where sqrtM                       = ceiling $ sqrt $ fromIntegral m
              lhsTable                    = [(powMod base i m, i) | i <- [ 1 .. (sqrtM - 1) ]]
              iterMap                     = [1 .. sqrtM]
              allSols                     = map (babyStepGiantStepPar' base x m sqrtM lhsTable) iterMap `using` parList
rpar
              allRight                    = filter isRight allSols

babyStepGiantStepPar' :: Integer -> Integer -> Integer -> Integer -> [(Integer, Integer)] -> Integer -> Either String
Integer
babyStepGiantStepPar' base x m sqrtM lhsTable numIter
  | elem rhsSol (map fst lhsTable) = Right ans
  | otherwise                      = Left "No match this iteration"
            where rhsSol           = x * (inverseEuclid (base ^ (currRhsIdx)) m) `mod` m
                  ans              = currLhsIdx + currRhsIdx
                  currLhsIdx       = getI rhsSol lhsTable
                  currRhsIdx       = sqrtM * numIter

-- if par with chunks
runBabyStepGiantStepParWithChunks :: String -> Either String Integer
runBabyStepGiantStepParWithChunks line =
    let [x, base, modulus] = map read $ words line
    in babyStepGiantStepParWithChunks base x modulus

babyStepGiantStepParWithChunks :: Integer -> Integer -> Integer -> Either String Integer
babyStepGiantStepParWithChunks base x m
  | isRelativelyPrime base m == False     = Left "Base and modulus must be relatively prime"
  | length allRight > 0                   = head allRight
  | otherwise                             = Left "No solution"
        where sqrtM                       = ceiling $ sqrt $ fromIntegral m
              lhsTable                    = [(powMod base i m, i) | i <- [ 1 .. (sqrtM - 1) ]]
              iterMap                     = chunksOf 50 [1 .. sqrtM]
              allSols                     = map (babyStepGiantStepParWithChunks' base x m sqrtM lhsTable) iterMap `using`
parList rpar
              allRight                    = filter isRight $ concat allSols

babyStepGiantStepParWithChunks' :: Integer -> Integer -> Integer -> Integer -> [(Integer, Integer)] -> [Integer] ->
[Either String Integer]
babyStepGiantStepParWithChunks' base x m sqrtM lhsTable iterMap =
    map (babyStepGiantStepPar' base x m sqrtM lhsTable) iterMap

-- if single
```

```haskell
runBabyStepGiantStep :: String -> Either String Integer
runBabyStepGiantStep line =
  let [x, base, modulus] = map read $ words line
  in babyStepGiantStep base x modulus

babyStepGiantStep :: Integer -> Integer -> Integer -> Either String Integer
babyStepGiantStep base x m
    | isRelativelyPrime base m == False = Left "Base and modulus must be relatively prime"
    | otherwise                         = babyStepGiantStep' base x m sqrtM 1 lhsTable
          where sqrtM                   = ceiling $ sqrt $ fromIntegral m
                lhsTable                = [(powMod base i m, i) | i <- [ 1 .. (sqrtM - 1) ]]

babyStepGiantStep' :: Integer -> Integer -> Integer -> Integer -> Integer -> [(Integer, Integer)] -> Either String
Integer
babyStepGiantStep' base x m sqrtM numIter lhsTable
    | elem rhsSol (map fst lhsTable) = Right ans
    | numIter > sqrtM                = Left "No solution"
    | otherwise                      = babyStepGiantStep' base x m sqrtM (numIter + 1) lhsTable
              where rhsSol           = x * (inverseEuclid (base ^ (currRhsIdx)) m) `mod` m
                    ans              = currLhsIdx + currRhsIdx
                    currLhsIdx       = getI rhsSol lhsTable
                    currRhsIdx       = sqrtM * numIter

-- HELPER FUNCTIONS --
powMod :: Integer -> Integer -> Integer -> Integer
powMod b e m = powMod' b e m 1

powMod' :: Integer -> Integer -> Integer -> Integer -> Integer
powMod' _ 0 _ result = result
powMod' b e m result = powMod' bNew eNew m resultNew
  where resultNew   = if (odd e) then (result * b) `mod` m else result
        eNew        = Bits.shift e (-1)
        bNew        = (b * b) `mod` m

inverseEuclid :: Integer -> Integer -> Integer
inverseEuclid x m = inverseEuclid' m x m 0 1 100

inverseEuclid' :: Integer -> Integer -> Integer -> Integer -> Integer -> Integer -> Integer
inverseEuclid' m x mUpdated a b c
  | c == 0        = a `mod` m
  | otherwise     = inverseEuclid' m newX x b y newX
      where newX = mUpdated `mod` x
            y    = a - (mUpdated `div` x) * b

isRelativelyPrime :: Integer -> Integer -> Bool
isRelativelyPrime num1 num2
  | num2 == 0 = (num1 == 1)
  | otherwise = isRelativelyPrime num2 (num1 `mod` num2)

getI :: Integer -> [(Integer, Integer)] -> Integer
getI word (hd:tl)
  | ((fst hd) == word) = snd hd
  | otherwise          = getI word tl
getI word []           = error "programming error!"
-- HELPER FUNCTIONS --
```

runRegex.hs

```
{-
param1 : input file to match regex on
param2 : regex to match words agaainst
return :  list of number of matches per line

$ ghc -O2 runRegex.hs -rtsopts -eventlog
$ ./runRegex sample_in.txt at
[1, 1]

sample_in.txt:
phosphatize Kristian pre-expound Kourou Asshur conquistadores Mayview Turkey-carpeted
Blessington xanthochroia cue Lamb basso-relievo diarize esthesioblast Natica


Regex that we support:
abc*        matches a string that has ab followed by zero or more c
a(bc)*      matches a string that has a followed by zero or more copies of the sequence bc
roar        matches any string that has the text roar in it
a(b|c)      matches a string that has a followed by b or c, multiple ors not supported
\d          matches a single character that is a digit
.           matches any character
-}

import Control.Parallel.Strategies
import System.Environment(getArgs)
import Regex

main :: IO ()
main = do
  [f, regex] <- getArgs
  file <- readFile f

  let line      = map words $ lines file
      sol       = map (runRegex regex) line
  putStrLn $ show $ map length sol
```

Regex.hs

```haskell
module Regex (runRegex) where

import Data.Char(isDigit, isAlphaNum)
import Data.List(isInfixOf, isPrefixOf)


runRegex :: String -> [String] -> [String]
runRegex regex contents =
    let res = map (matchRegex regex) contents
    in map (\(b, word) -> word) (filter (\(b, word) -> b) (zip res contents))

-- MATCH REGEX
-- basics
matchRegex :: String -> String -> Bool
matchRegex ('.':tlReg) (_:tlStr)     = matchRegex tlReg tlStr

-- or
matchRegex ('(':a:'|':b:')':tl) (hdStr:tlStr)
  | hdStr == a || hdStr == b         = matchRegex tl tlStr
  | otherwise                        = False
matchRegex ('(':_:'|':_:')':_) []    = False

-- kleene star
matchRegex regex@(a:'*':tl) (hdStr:tlStr)
  | a /= hdStr && tlStr == []        = False
  | a /= hdStr                       = matchRegex tl tlStr
  | a == hdStr                       = matchRegex tl tlStr || matchRegex regex tlStr

-- parenthesized kleene star
matchRegex regex@('(':_:_) str       = recurringKleene (regex) str []

-- escape characters
matchRegex ('\\':'d':tl) (hdStr:tlStr)
  | isDigit hdStr                    = matchRegex tl tlStr
  | otherwise                        = False

-- alphanum
matchRegex regex@(hdReg:tlReg) str@(hdStr:tlStr)
  | isAlphaNum' regex                = isInfixOf regex str
  | isAlphaNum hdReg && isAlphaNum hdStr && hdReg == hdStr
                                     = matchRegex tlReg tlStr
  | otherwise                        = False

matchRegex [] []                     = True
matchRegex [] _                      = False
matchRegex _ []                      = False
-- MATCH REGEX

-- HELPER FUNCTIONS
isAlphaNum' :: String -> Bool
isAlphaNum' (hd:tl)
  | isAlphaNum hd = isAlphaNum' tl
  | otherwise     = False
isAlphaNum' []    = True

removeKleene :: String -> String -> String
removeKleene kleene str = reverse (take ((length str) - (length $ tail kleene)) (reverse str))

recurringKleene :: String -> String -> String -> Bool
recurringKleene (')':'*':tl) str kleene
  | isPrefixOf (tail kleene) str
                       = (matchRegex (kleene ++ ")*" ++ tl) (removeKleene kleene str) || matchRegex tl str)
  | otherwise          = matchRegex tl str
recurringKleene (hd:tl) str kleene
```

```
                        = recurringKleene tl str (kleene ++ [hd])
recurringKleene [] _ _ = error "TODO how would this get called?"
-- HELPER FUNCTIONS
```