Parallel Functional Programing

Parallel Apriori Algorithm

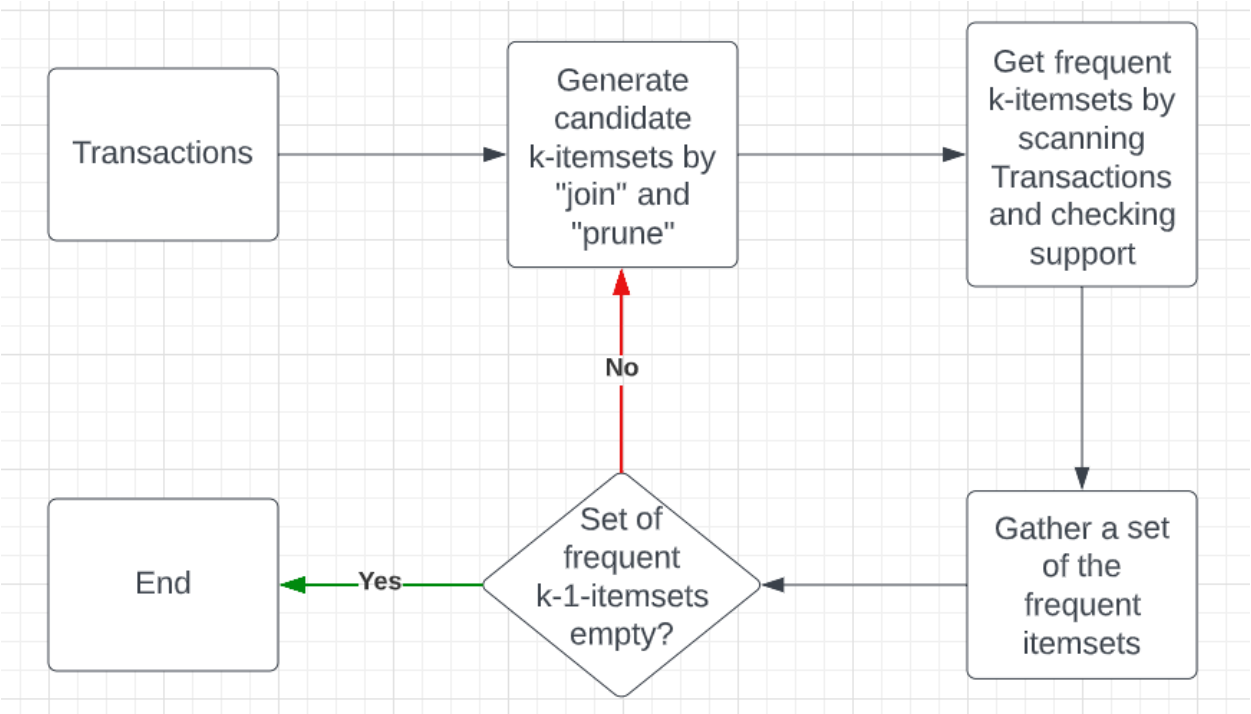
Yihan Yin (yy3114)

1. Introduction

This report mainly focuses on the Haskell implementation of Apriori Algorithm and some attempts to parallelize the sequential implementation, which managed to help improve the efficiency of the program. Some comparisons of the speedup between different cores are benchmarked for a better illustration.

2. Apriori Algorithm

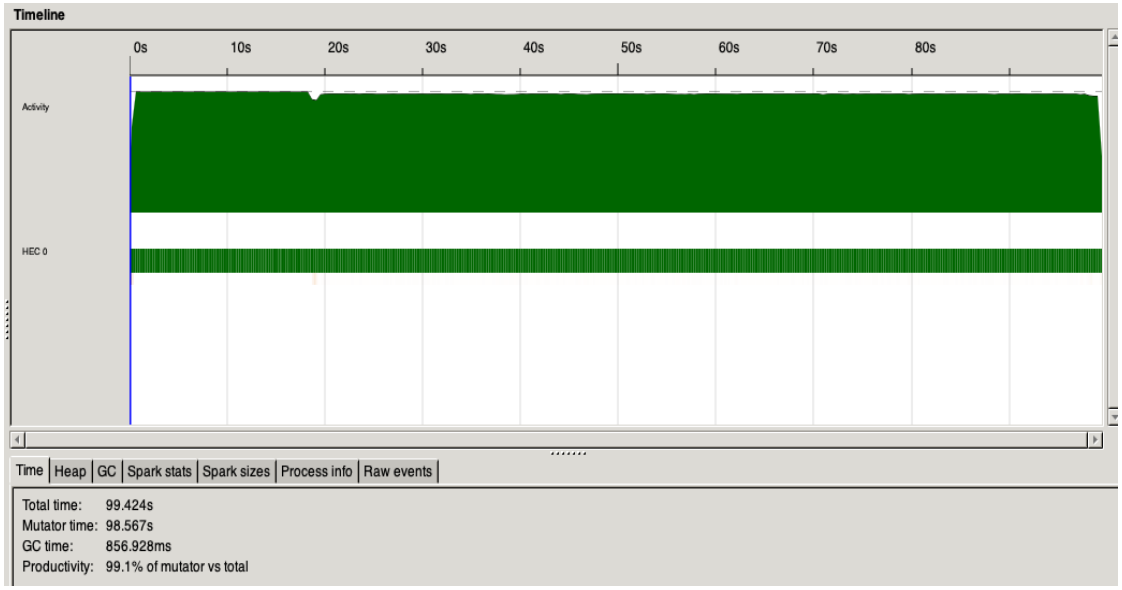
Apriori Algorithm is designed for frequent itemset mining and association rules learning over relational databases that contain transactions. This report mainly focuses on frequent itemsets mining. Itemset, as the name indicates, is a set of items. Transaction, can be seen as one itemset while itemset is not necessarily a transaction. Given a threshold, Apriori Algorithm is to identify frequent itemsets that are subsets of the transactions above the threshold. And the basic idea is as below:



1. Scan the transactions to get first itemsets whose support is larger than the threshold.

2. Based on the theory that if k-level itemset is a frequent itemset, all subsets of k-level itemset should also be a frequent itemset, we use “self join” to generate (k+1)-level itemset from k-level frequent itemset and then use “prune” to discard those non-frequent itemsets. Then we get (k+1)-level candidate itemsets.
3. From the (k+1)-level candidate itemsets, we check their support and only leave those larger than the support threshold. Then we get (k+1)-level frequent itemsets.
4. We then go back to 2, calculate the (k+2)-level candidate itemsets from (k+1)-level frequent itemsets. Then get (k+2)-level frequent itemsets from (k+2)-level candidate itemsets until our (k+n)-level frequent itemsets turn out empty.

   As a result, the implementation should be clear. The 3 important functions are **firstFreqItemsets, nextCandItemsets and nextFreqItemsets** where **firstFreqItemsets** is **step 1**, **nextCandItemsets** is about **step 2** and **nextFreqItemsets** represents **step 3**. Then the whole process makes up the Apriori Algorithm, which is exactly **step 4**.
   The figure below shows the eventlog of the sequential Apriori Algorithm Haskell implementation, with 10,000 transactions and 0.5% as the minimal support threshold.

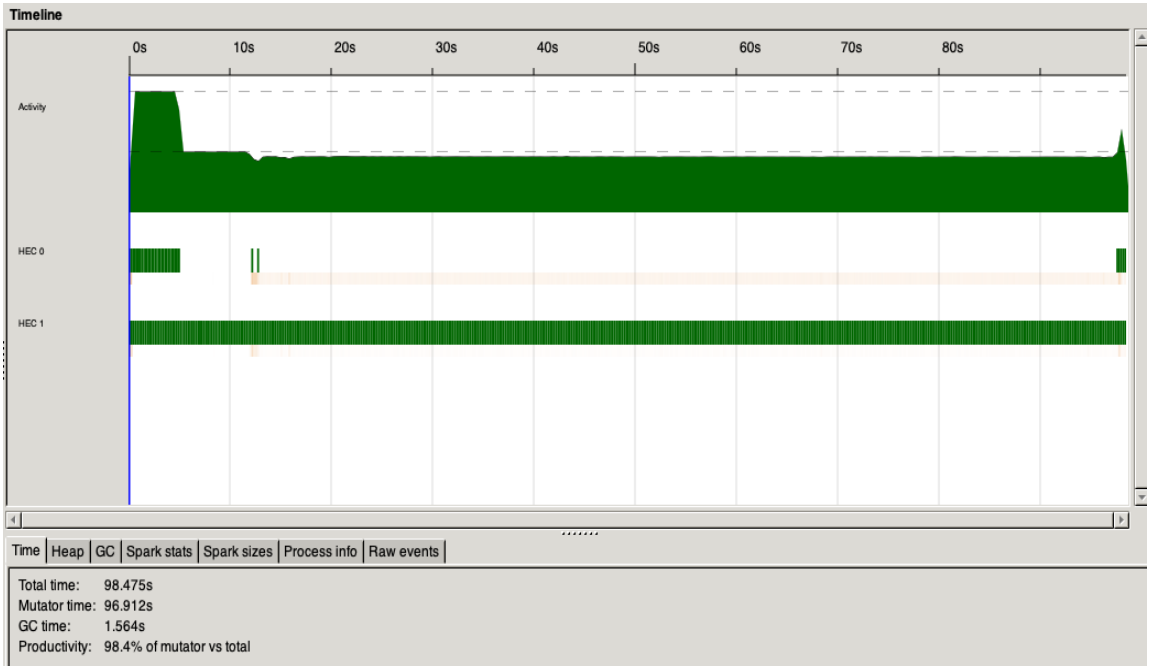


3. Parallel Implementation

As discussed in Section 2, the implementation has 3 important functions. With closer look, it can be found that every itemset in its k-level itemsets, collection of itemset, is independent of each other. To be specific, calculating the support of each 1-level itemset in **firstFreqItemsets** won’t be affected by other 1-level itemset; generating the next level candidate itemset is also independent as the every k-level frequent itemset can possibly generate a (k+1)-level candidate itemset; Same reasons apply to generating (k+1)-level frequent itemsets.

parList

Turning to Control.Parallel.Strategies, since each element in the collection is independent, we can calculate each element in the collection in parallel. As a result, my first attempt was to apply **parList** to each 3 functions. And the eventlog is as below, still tested on 10,000 transactions with 0.5% minimal support threshold:

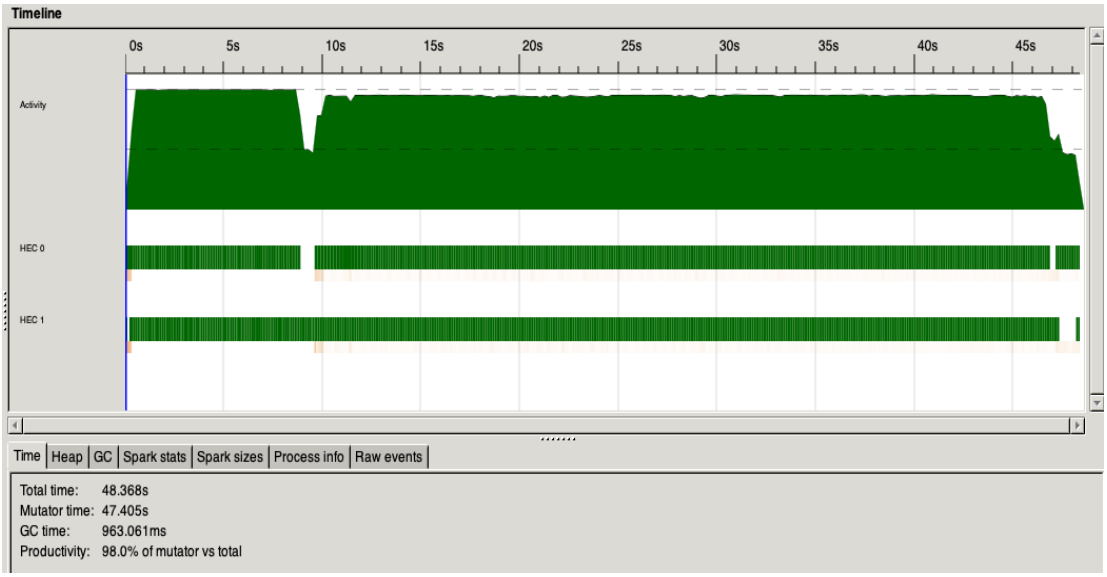


It can be found that it is not even optimized as the overhead is slowing things down, chunks of 1 lead to a fairly great amount of time for a spark to spawn, and thus give too much overhead. In the end, the time consumed is pretty much the same as the sequential one.

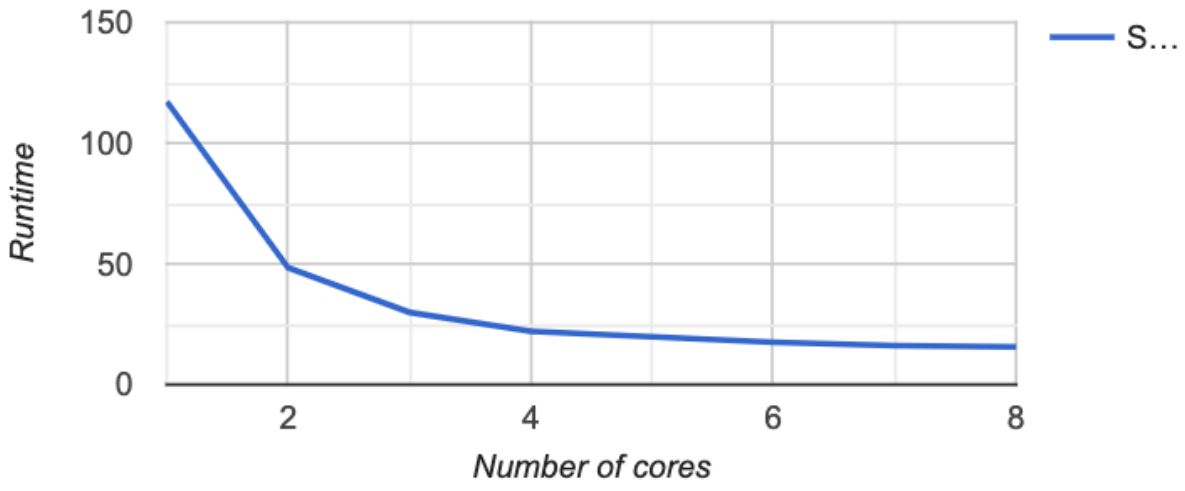
parListChunk

Reflecting on the overhead resulting from small size chunks, I made my second attempt to apply **parListChunk** so that the strategy is creating a spark for each chunk of a certain size in the collection, other than every element of them. And the eventlog is

shown below, with 500 as the chunk size:



And the speed turns out almost doubled compared to the sequential implementation. To further analyze the performance, I also tested the same program with different numbers of cores. The following two charts show the runtime comparison and the speedup respectively:

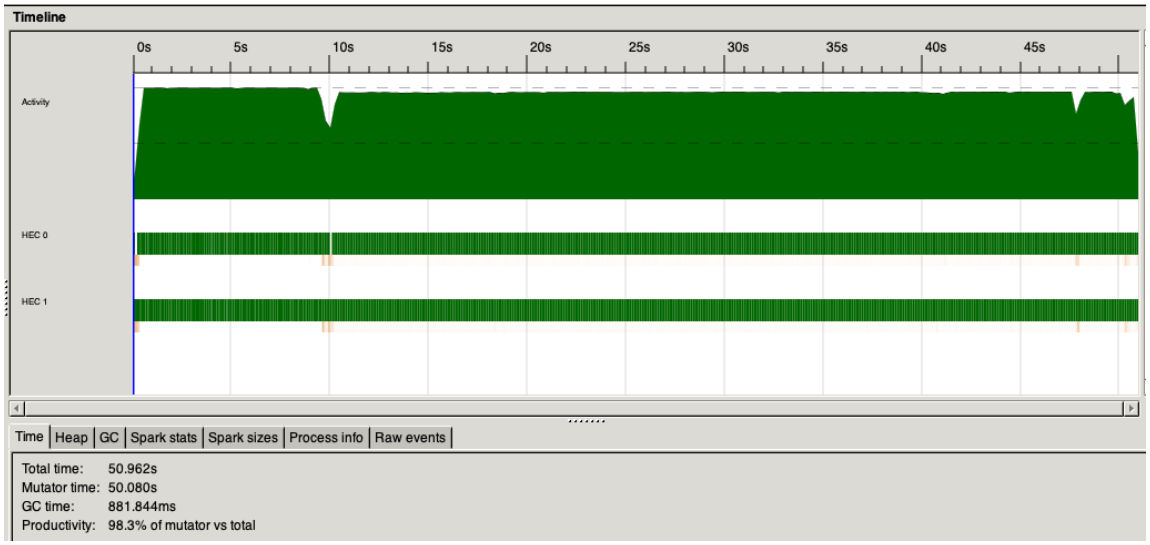


	Runtime	Speedup
1	117.109	1.00
2	48.368	2.42
3	29.903	3.91
4	22.015	5.32
5	19.781	5.92
6	17.592	6.65
7	16.13	7.26
8	15.636	7.49

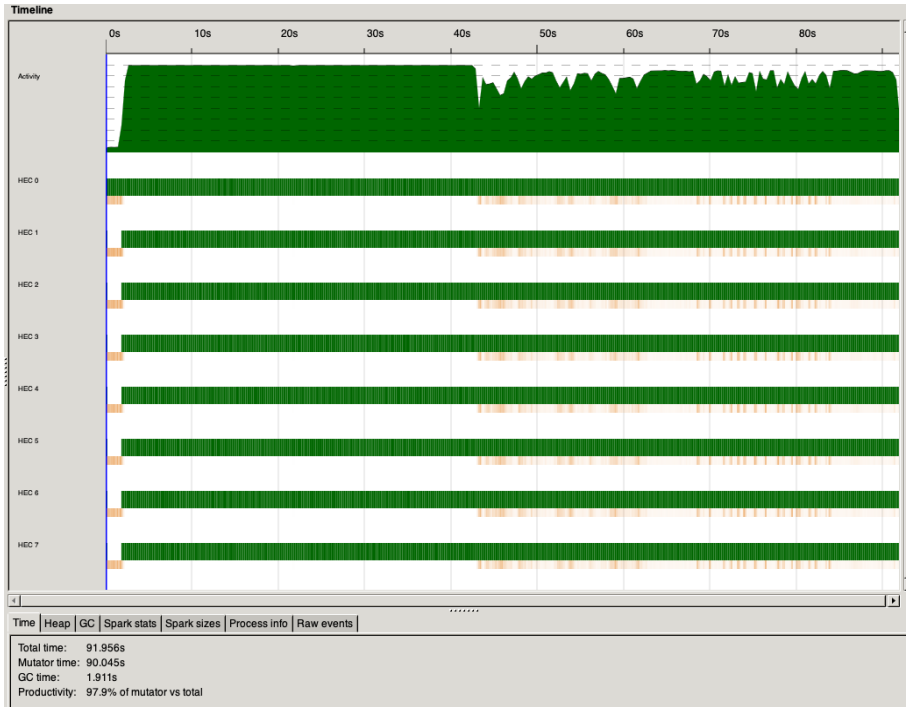
Before 4 cores, the performance improvement is quite great while when it's running on more than 4 cores, the improvement is not that impressive.

parMap

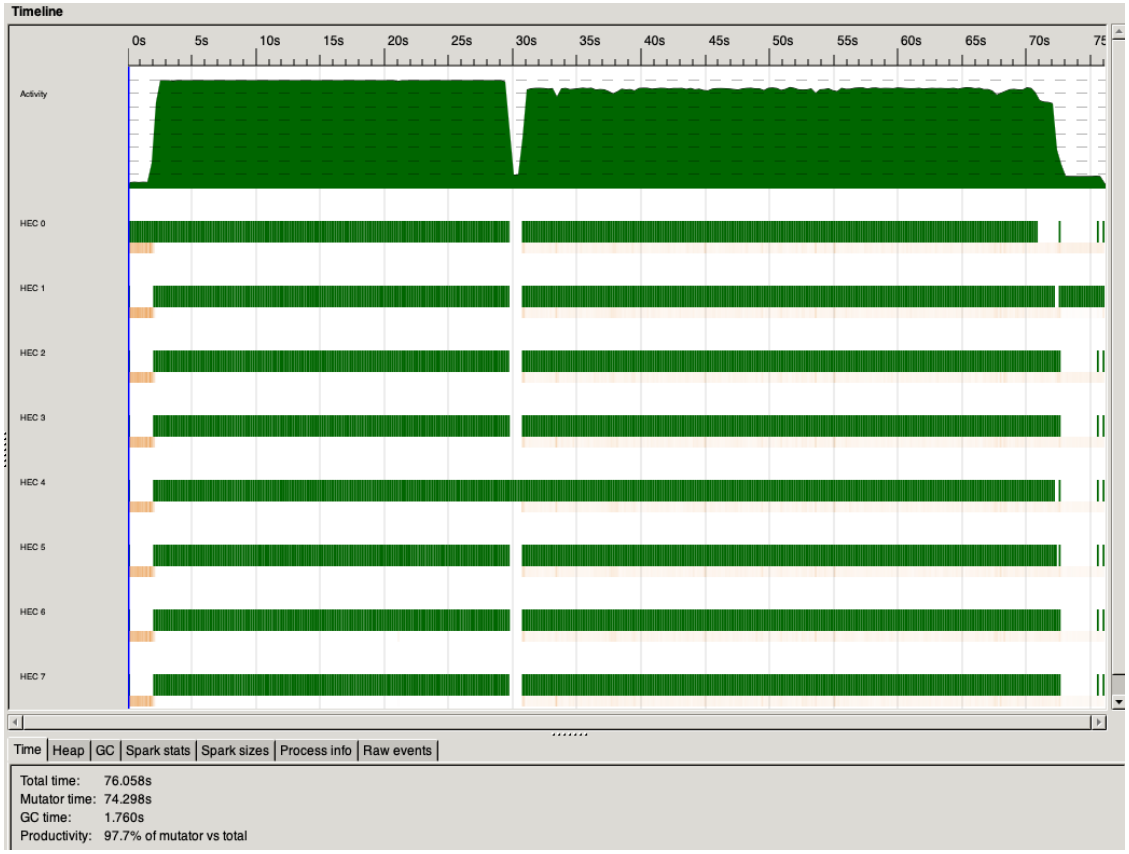
This attempt introduces Control.Monad.Par. **parMap** function applies the given function to each element of a data structure in parallel. That is, fully evaluating the results, and returning a new data structure containing the results. It seems pretty similar with **parList** in Control.Parallel.Strategies, as they both handle every element of a collection. However, the efficiency turns out to be much different on the same 10,000 transactions and 0.5% minimal support threshold:



It might be because Par Monad avoids the laziness issues and helps productive parallel programming. And the following eventlog shows implementation applying **parMap** running on 8 cores, with 50,000 transactions and 0.5% minimal support threshold, and average runtime is around 90 seconds:



And it performs pretty well when it comes to a larger dataset while **parListChunk** of 500 chunk size performs better than this one, with average runtime around 75 seconds:



4. Conclusion

This report includes both sequential and parallel versions of the Apriori Algorithm in Haskell and analyzes different attempts about the parallelism, and the performance comparison on different numbers of cores. **parListChunk** with different strategies does give impressive performance improvement but the chunk size has to be experimented to figure out. With **parMap** in Control.Monad.Par, one doesn't need to try out the chunk size but has to take the risk of fully evaluating results, for example, **parMap** over records from a huge file.

References

https://dwgeek.com/mining-frequent-itemsets-apriori-algorithm.html/
https://gist.github.com/cs/2909095
http://www.cs.columbia.edu/~sedwards/classes/2021/4995-fall/reports/Apriori.pdf
https://stackoverflow.com/questions/23326920/difference-between-par-monad-and-eval-monad-with-deepseq

Code

Main.hs

```
module Main (main) where

import Apriori (Itemset(..), Support, nextFreqItemsets, firstFreqItemsets,
parNextFreqItemsets, parFirstFreqItemsets, monadParNextFreqItemsets,
monadParFirstFreqItemsets)

import System.Exit (die)
import System.Environment (getArgs)
import qualified Data.List as List
import qualified Data.Set as Set


wordsWhen :: (Char -> Bool) -> String -> [String]
wordsWhen p s =
 case dropWhile p s of
   "" -> []
   s' -> w : wordsWhen p s''
       where (w, s'') = break p s'

getTransactions :: String -> [Itemset]
getTransactions content =
 map getItemset (lines content)
   where
     getItemset line = Itemset $ Set.fromList (wordsWhen (==',') line)

main :: IO()
main = do
 args <- getArgs
 case args of
   [file, supp] -> do
     let minSupp = read supp :: Support
     content <- readFile file
     let transactions = getTransactions content
     let freqItemsets = concat $ List.unfoldr (nextFreqItemsets minSupp
transactions) (firstFreqItemsets minSupp transactions)
     print $ freqItemsets
   [file, supp, "parallel", n] -> do
     let minSupp = read supp :: Support
```

```
    let chunk = read n :: Int
    content <- readFile file
    let transactions = getTransactions content
    let freqItemsets = concat $ List.unfoldr (parNextFreqItemsets chunk
minSupp transactions) (parFirstFreqItemsets chunk minSupp transactions)
    print $ freqItemsets
  [file, supp, "monad-par"] -> do
    let minSupp = read supp :: Support
    content <- readFile file
    let transactions = getTransactions content
    let freqItemsets = concat $ List.unfoldr (monadParNextFreqItemsets
minSupp transactions) (monadParFirstFreqItemsets minSupp transactions)
    print $ freqItemsets
  _ -> do
    die "IllegalArgumentException: should be {filename, minSupport,
['parallel', listChunkSize] | ['monad-par']}"
```

Apriori.hs

```
module Apriori where

import qualified Data.Set as Set
import Control.Monad.Par (parMap, runPar)
import Control.DeepSeq
import Control.Parallel.Strategies (using, parListChunk, rdeepseq)

-- definitions
type Support = Double

data Itemset = Itemset (Set.Set String) deriving (Eq, Ord)
instance Show Itemset where
 show (Itemset i) = show $ Set.toList i
instance NFData Itemset where
 rnf (Itemset i) = rnf i

-- calculate support
supportCount :: [Itemset] -> Itemset -> Int
supportCount transactions (Itemset i) =
 length $ filter (Set.isSubsetOf i) $ map (\(Itemset x) -> x) transactions

support :: [Itemset] -> Itemset -> Support
support transactions is =
 fromIntegral (supportCount transactions is) / fromIntegral (length transactions)
```

```haskell
-- util
deduplicate :: Ord a => [a] -> [a]
deduplicate l = Set.toList $ Set.fromList l

allSubsets :: Set.Set a -> [Set.Set a]
allSubsets s = Set.toList $ Set.powerSet s

allMaximalProperSubsets :: Set.Set a -> [Set.Set a]
allMaximalProperSubsets s = filter (\x -> (Set.size x) == (Set.size s) - 1) $
allSubsets s

-- apriori
firstCandItemsets :: [Itemset] -> [Itemset]
firstCandItemsets transactions =
 deduplicate $ concatMap (\(Itemset is) -> map (Itemset . Set.singleton) $
Set.toList is) transactions

firstFreqItemsets :: Support -> [Itemset] -> [Itemset]
firstFreqItemsets minSupp transactions =
 filter (\candIs -> support transactions candIs > minSupp) (firstCandItemsets
transactions)

nextCandItemsets :: [Itemset] -> [Itemset]
nextCandItemsets itemsets =
 -- two k-1 sets should differ in exactly 1 element before joining
 let validate a b = (Set.size $ a `Set.difference` b) == 1 in
 -- self join
 let nextIss = [Itemset (a `Set.union` b) | (Itemset a) <- itemsets, (Itemset b) <-
itemsets, validate a b] in
 -- prune
 let subsetsFrequent (Itemset is) = all (\s -> (Itemset s) `elem` itemsets)
(allMaximalProperSubsets is) in
 deduplicate $ filter subsetsFrequent nextIss

nextFreqItemsets :: Support -> [Itemset] -> [Itemset] -> Maybe ([Itemset],
[Itemset])
nextFreqItemsets _ _ [] = Nothing
nextFreqItemsets minSupp transactions curr =
 Just (curr, next)
   where
     next = filter (\candIs -> support transactions candIs > minSupp)
(nextCandItemsets curr)

-- parallel pkg
parFirstFreqItemsets :: Int -> Support -> [Itemset] -> [Itemset]
parFirstFreqItemsets n minSupp transactions =
```

```haskell
 let fullFirstIss = map (\candIs -> (candIs, support transactions candIs >
minSupp)) (firstCandItemsets transactions) `using` parListChunk n rdeepseq in
 concat $ [[candIs] | (candIs, isGood) <- fullFirstIss, isGood]

parNextCandItemsets :: Int -> [Itemset] -> [Itemset]
parNextCandItemsets n itemsets =
 -- two k-1 sets should differ in exactly 1 element before joining
 let validate a b = (Set.size $ a `Set.difference` b) == 1 in
 -- self join
 let nextIss = [Itemset (a `Set.union` b) | (Itemset a) <- itemsets, (Itemset b) <-
itemsets, validate a b] in
 -- prune
 let subsetsFrequent (Itemset is) = all (\s -> (Itemset s) `elem` itemsets)
(allMaximalProperSubsets is) in
 -- get full next itemsets [(is, good)] in parallel
 let fullNextIss = map (\nextIs -> (nextIs, subsetsFrequent nextIs)) nextIss
`using` parListChunk n rdeepseq in
 deduplicate $ concat $ [[nextIs] | (nextIs, isGood) <- fullNextIss, isGood]

parNextFreqItemsets :: Int -> Support -> [Itemset] -> [Itemset] -> Maybe
([Itemset], [Itemset])
parNextFreqItemsets _ _ _ [] = Nothing
parNextFreqItemsets n minSupp transactions curr =
 let fullNextCandIss = map (\candIs -> (candIs, support transactions candIs >
minSupp)) (parNextCandItemsets n curr) `using` parListChunk n rdeepseq in
 let next = concat $ [[candIs] | (candIs, isGood) <- fullNextCandIss, isGood] in
 Just (curr, next)

-- monad-parallel
monadParFirstFreqItemsets :: Support -> [Itemset] -> [Itemset]
monadParFirstFreqItemsets minSupp transactions =
 let fullFirstIss = runPar $ parMap (\candIs -> (candIs, support transactions
candIs > minSupp)) (firstCandItemsets transactions) in
 concat $ [[candIs] | (candIs, isGood) <- fullFirstIss, isGood]

monadParNextCandItemsets :: [Itemset] -> [Itemset]
monadParNextCandItemsets itemsets =
 -- two k-1 sets should differ in exactly 1 element before joining
 let validate a b = (Set.size $ a `Set.difference` b) == 1 in
 -- self join
 let nextIss = [Itemset (a `Set.union` b) | (Itemset a) <- itemsets, (Itemset b) <-
itemsets, validate a b] in
 -- prune
 let subsetsFrequent (Itemset is) = all (\s -> (Itemset s) `elem` itemsets)
(allMaximalProperSubsets is) in
 -- get full next itemsets [(is, good)] in parallel
 let fullNextIss = runPar $ parMap (\nextIs -> (nextIs, subsetsFrequent nextIs))
```

```
nextIss in
 deduplicate $ concat $ [[nextIs] | (nextIs, isGood) <- fullNextIss, isGood]

monadParNextFreqItemsets :: Support -> [Itemset] -> [Itemset] -> Maybe ([Itemset],
[Itemset])
monadParNextFreqItemsets _ _ [] = Nothing
monadParNextFreqItemsets minSupp transactions curr =
 let fullNextCandIss = runPar $ parMap (\candIs -> (candIs, support transactions
candIs > minSupp)) (monadParNextCandItemsets curr) in
 let next = concat $ [[candIs] | (candIs, isGood) <- fullNextCandIss, isGood] in
 Just (curr, next)
```