The Parallel Apriori Algorithm [*]

Hongfei Chen (hc3222)

December 23, 2021

1 Introduction

The report describes a parallel Haskell implementation of the Apriori Algorithm from the paper "Fast algorithms for mining association rules" (Agrawal & Srikant, 1994). Section 2 includes an overview of the Apriori Algorithm as well as the sequential implementation in Haskell. Section 3 introduces two layers of parallelism being applied to the sequential implementation, which significantly improves the performance of the Apriori Algorithm.

2 Apriori Algorithm

2.1 Overview

The Apriori Algorithm is an algorithm for data mining, in particular, association rule mining. It searches for boolean association rule of the frequent itemsets in a dataset, which is useful for discovering the items that tend to appear together in a transaction.

The main idea of the algorithm is that for every possible size of itemsets, generate the candidate frequent itemsets from the smaller-sized frequent itemsets and then filter the candidates based on the required minimum support value (Figure 1). The candidate generation consists of the join step (Figure 2) and the prune step (Figure 3), in which the algorithm finds the candidate size-k itemsets by self-joining the size-$(k-1)$ itemsets, and then prune those who have a subset which is not a size-$(k-1)$ itemset. Finally, the association rules which satisfy the minimum confidence value will be output.

```
1)  $L_1$ = {large 1-itemsets};
2)  for ( $k = 2$; $L_{k-1} \neq \emptyset$; $k$++ ) do begin
3)      $C_k$ = apriori-gen($L_{k-1}$); // New candidates
4)      forall transactions $t \in \mathcal{D}$ do begin
5)          $C_t$ = subset($C_k$, $t$); // Candidates contained in $t$
6)          forall candidates $c \in C_t$ do
7)              $c$.count++;
8)      end
9)      $L_k = \{c \in C_k \mid c.\text{count} \geq \text{minsup}\}$
10) end
11) Answer = $\bigcup_k L_k$;
```

Figure 1: The Apriori Algorithm

[*] This is the final project report for COMS 4995 Parallel Functional Programming, Fall 2021.

```
insert into $C_k$
select $p.\text{item}_1$, $p.\text{item}_2$, ..., $p.\text{item}_{k-1}$, $q.\text{item}_{k-1}$
from $L_{k-1}$ $p$, $L_{k-1}$ $q$
where $p.\text{item}_1 = q.\text{item}_1$, . . ., $p.\text{item}_{k-2} = q.\text{item}_{k-2}$,
        $p.\text{item}_{k-1} < q.\text{item}_{k-1}$;
```

Figure 2: `apriori-gen` Join Step

```
forall itemsets $c \in C_k$ do
   forall $(k-1)$-subsets $s$ of $c$ do
      if $(s \notin L_{k-1})$ then
         delete $c$ from $C_k$;
```

Figure 3: `apriori-gen` Prune Step

2.2 Haskell Implementaion

The program takes in three arguments, which are the filename of CSV dataset, the minimum support value and the minimum confidence value, i.e. `stack exec apriori-parallel-exe <csv_filename> <min_support> <min_confidence>`. More usage information can be found in Appendix A and the README file. The output of the program is a set of association rules with corresponding support and confidence value. The test datasets provided in the code zip file are retail transaction datasets (Carrie, 2018). Hence, the association rule in this context is that, if a customer bought product A, how likely they would buy product B, where the likelihood is indicated by the confidence value. The support value, on the other hand, describes how frequent an itemset appear in the transactions.

While the complete implementation can be found in Appendix B, this section will demonstrate parts of the code that contain the essential steps of the Apriori Algorithm. They are also significant to the parallelism implementation introduced in the next section.

The algorithm starts with the initial size-1 frequent itemsets, which are generated directly from the input transactions.

```haskell
getInitFreqItemset :: Double -> [Itemset] -> [Itemset]
getInitFreqItemset minSupport transactions =
    let initCandItemset = removeDup $
            concatMap (\(Itemset t) -> map (Itemset . Set.singleton) $ Set.toList t) transactions
    in filter (\cand -> getSupport transactions cand > minSupport) initCandItemset
```

Then, using the size-1 frequent itemsets as input, the "getFreqItemsets" function recursively generates larger candidate itemsets through the aprioriGen function, and filter out those don't have enough support.

```haskell
getFreqItemsets :: Double -> [Itemset] -> [Itemset] -> Maybe ([Itemset], [Itemset])
getFreqItemsets _ _ [] = Nothing
getFreqItemsets minSupport transactions currFreqItemset =
    let nextCandItemset = aprioriGen currFreqItemset
        nextFreqItemset = filter (\cand -> getSupport transactions cand > minSupport) nextCandItemset
    in Just (currFreqItemset, nextFreqItemset)
```

In the aprioriGen function, it first does a self-join and then prune the results to make sure that, for a size-k itemset, all its size-(k-1) subsets are in the frequent itemsets generated in the previous iteration.

```haskell
aprioriGen :: [Itemset] -> [Itemset]
aprioriGen iss =
    let
        -- join step
        selfJoin = [Itemset (a `Set.union` b)
            | (Itemset a) <- iss, (Itemset b) <- iss, validateCandidate a b]
        validateCandidate a b = Set.size (a `Set.difference` b) == 1
        -- prune step
        nonFrequentSubsets (Itemset i) = all (\s -> Itemset s `elem` iss) (properSubsets i)
        powerSetList s = Set.toList $ Set.powerSet s
        properSubsets s = filter (\x -> Set.size x == Set.size s - 1) (powerSetList s)
        candItemset = filter nonFrequentSubsets selfJoin
    in removeDup candItemset
```

Figure 4 shows an example eventlog for running the sequential program on a test dataset containing 2000 transactions, with the minimum support value set to 0.1% and 50% minimum confidence. The average runtime for the sequential implementation is about 40.32 seconds. As this is the baseline of the program performance, the tests run in the next section are all using this set of input parameters, unless otherwise specified.

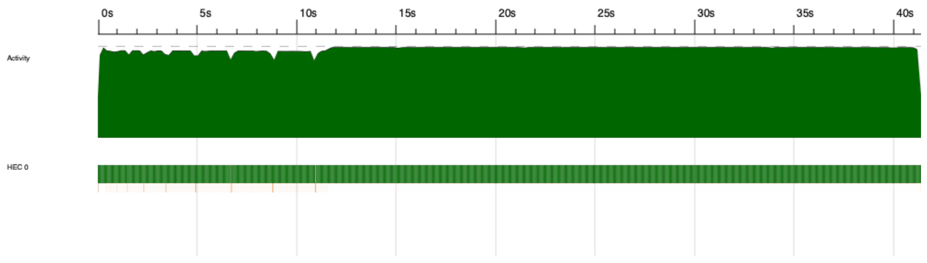


Figure 4: Sequential program eventlog
(2000 transactions, 0.1% support, 50% confidence)

3 Parallelism

There are two layers of parallelism applied to the implementation to improve the performance of the program. The first and inner layer uses the `parMap` function from the package `Control.Monad.Par`. The second and outer layer adopts the idea of MapReduce.

3.1 Par Monad

The `parMap` function applies a given function to each element in the list in parallel, fully evaluates them and return the results. My first attempt was to apply `parMap` in the `getFreqItemsets` function when it is filtering the itemset without enough support value. Since this is a filtering process instead of simple mapping, the results returned by `parMap` need to be concatenated.

```
getFreqItemsets :: Double -> [Itemset] -> [Itemset] -> Maybe ([Itemset], [Itemset])
getFreqItemsets _ _ [] = Nothing
getFreqItemsets minSupport transactions currFreqItemset =
    let nextCandItemset = aprioriGen currFreqItemset
        --- parMap
        nextFreqItemsetLst = runPar $
            parMap (\cand -> (cand, getSupport transactions cand > minSupport)) nextCandItemset
        nextFreqItemset = concat $ [[cand] | (cand, isFreq) <- nextFreqItemsetLst, isFreq]
    in Just (currFreqItemset, nextFreqItemset)
```

By converting sequential filtering procedure to a parallel one, the runtime of the program gets reduced by about 10 seconds. Despite the performance improvement, the eventlog (Figure 5) shows that apparently the parallelism is only taking effect on the second half of the program.

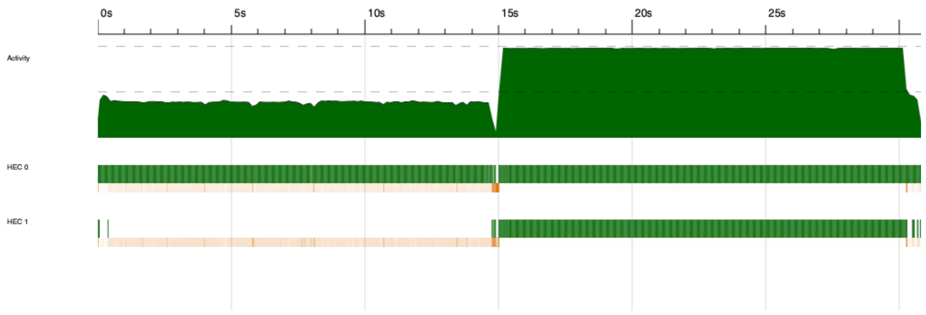


Figure 5: Parallelism with parMap eventlog 1
(2000 transactions, 0.1% support, 50% confidence)

Therefore, I apply the same parallelism logic to the `getInitFreqItemset` function, since the process of `getInitFreqItemset` is analogous to that of `getFreqItemsets`.

```
getInitFreqItemset :: Double -> [Itemset] -> [Itemset]
getInitFreqItemset minSupport transactions =
    let initCandItemset = removeDup $
            concatMap (\(Itemset t) -> map (Itemset . Set.singleton) $ Set.toList t) transactions
    --- parMap
        initFreqItemset = runPar $
            parMap (\cand -> (cand, getSupport transactions cand > minSupport)) initCandItemset
    in concat $ [[cand] | (cand, isFreq) <- initFreqItemset, isFreq]
```

However, as the `getInitFreqItemset` function will only be called once at the initial step and it is not computationally heavy, making it parallel has very limited impact on the program performance- on average, it reduces the runtime by 2 seconds. As shown in Figure 6, there still remains a rather large chunk of sequential process.

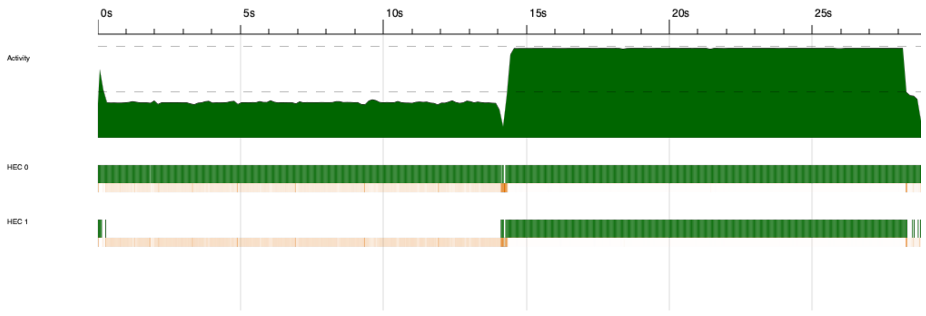


Figure 6: Parallelism with parMap eventlog 2
(2000 transactions, 0.1% support, 50% confidence)

Given the results above, the sequential part appears to happen within the `aprioriGen` function. Notice that the prune step of `aprioriGen` is performance pretty heavy computation, as it needs to check for every size-$(k-1)$ subset for a long list of itemsets returned by the join step. Hence, I added another layer of parallelism to the prune step.

```
aprioriGen :: [Itemset] -> [Itemset]
aprioriGen iss =
    let
        -- join step
        selfJoin = [Itemset (a `Set.union` b)
            | (Itemset a) <- iss, (Itemset b) <- iss, validateCandidate a b]
        validateCandidate a b = Set.size (a `Set.difference` b) == 1
        -- prune step
        nonFrequentSubsets (Itemset i) = all (\s -> Itemset s `elem` iss) (properSubsets i)
        powerSetList s = Set.toList $ Set.powerSet s
        properSubsets s = filter (\x -> Set.size x == Set.size s - 1) (powerSetList s)
        --- parMap
        candItemsetLst = runPar $ parMap (\cand -> (cand, nonFrequentSubsets cand)) selfJoin
        candItemset = concat $ [[cand] | (cand, isSubSet) <- candItemsetLst, isSubSet]
    in removeDup candItemset
```

By adding parallelism to all these three functions, the performance of the resulting the program has reduced to about 23 seconds, which is almost a half of the 40-second sequential program. From the evenlog (Figure 7), the implementation renders good parallelism balancing running on 2 cores.

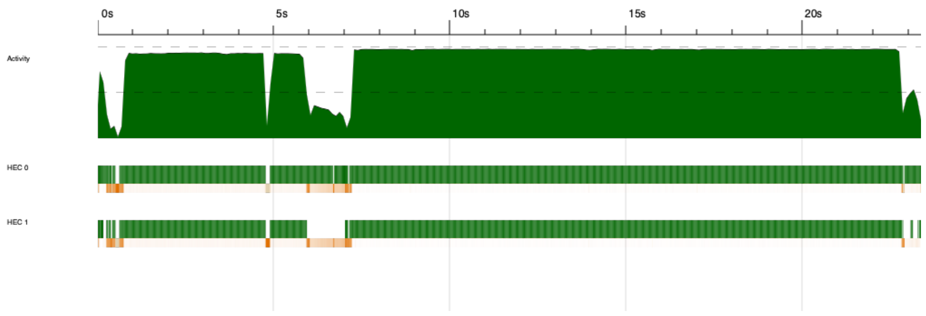


Figure 7: Parallelism with parMap eventlog 3
(2000 transactions, 0.1% support, 50% confidence)

To further analyze the performance, I ran the tests with the same input parameters on different number of cores. The graph in Figure 8 and the speedup comparison table in Figure 9 demonstrate the respective performance. The increase in the number cores results in better performance at first, but when running on more 4 cores, the return starts diminishing and the effect is no longer significant.

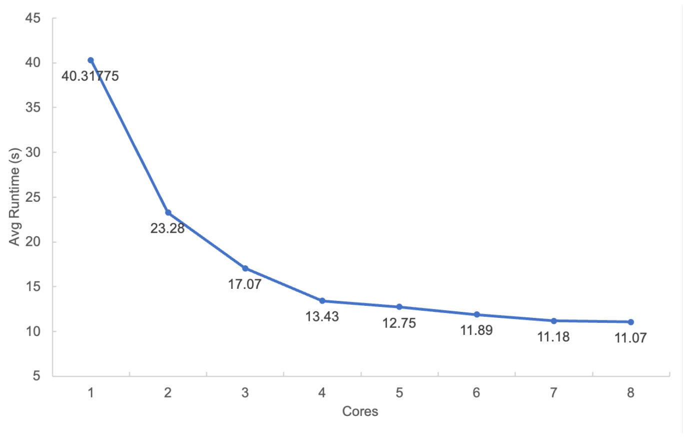
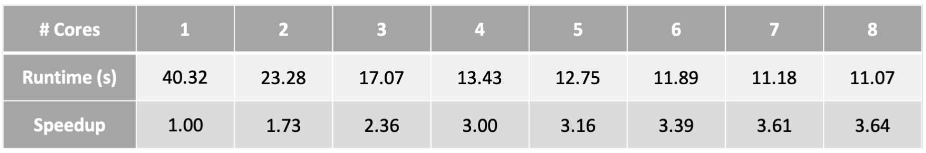


Figure 8: Average runtime over number of cores
(2000 transactions, 0.1% support, 50% confidence)



# Cores	1	2	3	4	5	6	7	8
Runtime (s)	40.32	23.28	17.07	13.43	12.75	11.89	11.18	11.07
Speedup	1.00	1.73	2.36	3.00	3.16	3.39	3.61	3.64

Figure 9: Average speedup over number of cores
(2000 transactions, 0.1% support, 50% confidence)

Figure 10 shows the evenlog when running the parallelized program on 8 cores. While the overall balancing is good, there remains some parts that run sequentially. The sequential chunk in the middle part is particularly outstanding. Since the parallel implementation discussed above involves mapping the list in parallel and then concatenate each element returned by the mapping, while the mapping is evaluated in parallel, the concatenation part is done sequentially. Hence, for the `aprioriGen` function where there tends to be a large number of itemsets to be concatenated, the resulting sequential process becomes non-negligible. Therefore, in the next section, I will explore ways to further improve the performance by making the program running on another layer of parallelism, so that the sequential part here can be run in parallel as well.

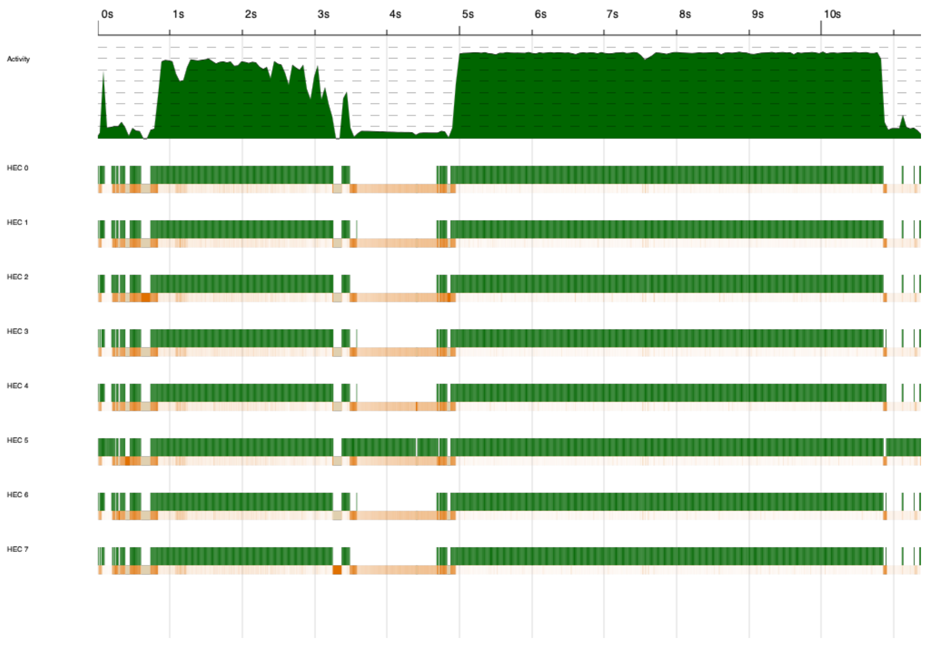


Figure 10: Parallelism with parMap eventlog 4
(2000 transactions, 0.1% support, 50% confidence)

3.2 MapReduce

In addition to applying parallelisms within the Apriori algorithm, I added another layer of parallelism so that the program will run the Apriori algorithm in parallel on top of parallel implementation discussed in the previous section, which is essentially applying the idea of MapReduce.

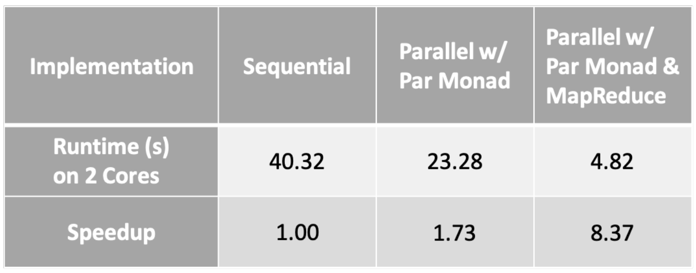
My initial attempt was to divide the transactions from the input dataset into smaller chunks and apply the Apriori algorithm to each chunk using the `parBuffer 100 rdeepseq` strategy. For the strategy to work, I also added the `NFData` instances for two data types I defined. The `aprioriByChunkSize` function takes a parameter indicating the size of each chunk, i.e. the number of transactions in each chunk.

```
newtype Itemset = Itemset (Set.Set String) deriving (Eq, Ord)
instance NFData Itemset where
  rnf (Itemset i) = rnf i

data AssocRule = AssocRule (Set.Set String) (Set.Set String) Double Double deriving (Eq, Ord)
instance NFData AssocRule where
    rnf (AssocRule a b s c) = rnf a `seq` rnf b `seq` rnf s `seq` rnf c


aprioriByChunkSize :: Int -> [Itemset] -> Double -> Double -> [AssocRule]
aprioriByChunkSize n transactions support confidence =
    removeDup (concatMap (\c -> apriori support confidence c transactions) chunks
            `using` parBuffer 100 rdeepseq)
        where
            chunk _ [] = []
            chunk n xs = let (as,bs) = splitAt n xs in as : chunk n bs
            chunks = chunk n transactions
```

The comparison table in Figure 11 shows a drastic speedup with the additional layer of parallelism using the MapReduce method. Figure 12 is a sample eventlog for running the MapReduce parallelism on 2 cores with chunk size 5. The balancing of the program is significantly better than earlier, as there is essentially no more sequential part when the program is running.



Implementation	Sequential	Parallel w/ Par Monad	Parallel w/ Par Monad & MapReduce
Runtime (s) on 2 Cores	40.32	23.28	4.82
Speedup	1.00	1.73	8.37

Figure 11: Performance comparison
(2000 transactions, 0.1% support, 50% confidence, chunk size 5)

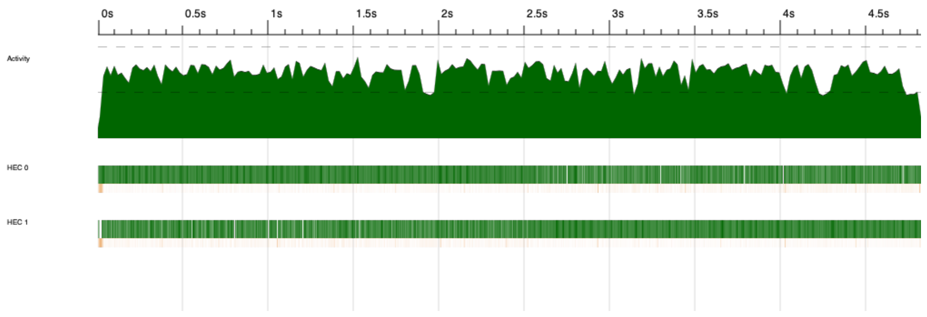


Figure 12: Parallel with MapReduce eventlog

Next, I altered the chunk size to see if there is an optimal size that renders the best performance. Figure 13 demonstrates the relations on the chunk size, core number and the runtime. The figure infers that the best performance is achieved when the chunk size is set to 1, regardless of the number of cores. Further, with chunk size 1, there is no noticeable runtime difference between running on different number of cores. A potential factor that might leads to negligible effect of chunk size is the size of the input transactions. As the tests so far are run on 2000 transactions. To test the hypothesis, I ran more tests on a larger dataset consisting of 9000 transactions running on 4 cores. However, as shown in Figure 14, even when running on a larger dataset, the chunk size is still optimal at size 1. Given this observation, modify the code to eliminate the divide-into-chunk part, and simply do a parallel mapping on each transaction.

```haskell
aprioriChunk :: [Itemset] -> Double -> Double -> [AssocRule]
aprioriChunk transactions support confidence =
    removeDup (concatMap (\c -> apriori support confidence [c] transactions) transactions
                `using` parBuffer 100 rdeepseq)
```

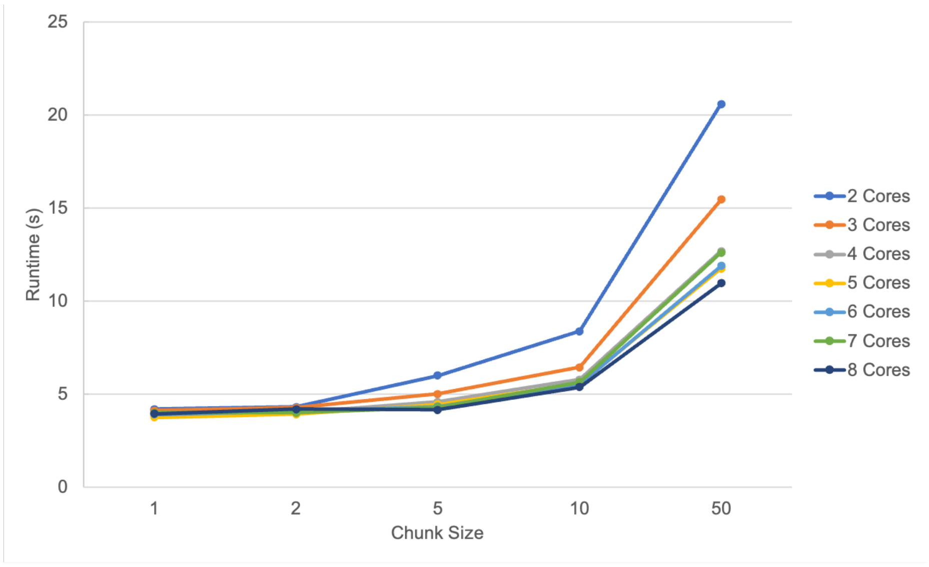


Figure 13: Runtime over chunk size and core number

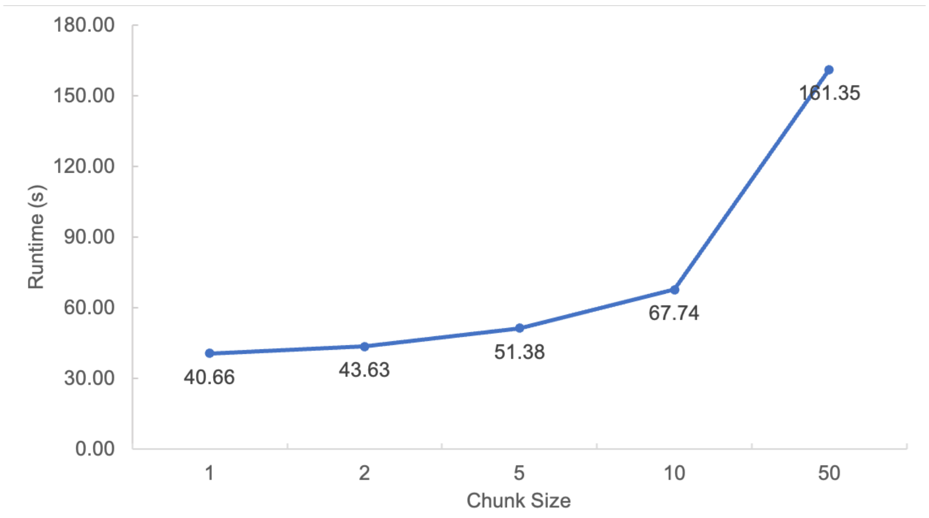


Figure 14: Runtime over chunk size on 4 cores
(9000 transactions, 0.1% support, 50% confidence)

Another aspect to be explored is the reason why the number of cores does not have an effect on the runtime. Looking at the output of the program, the largest itemset is of size 3 and there are only few of them. By the logic of the Apriori algorithm, a larger output itemset necessarily means more rounds of iterations, which leads to longer process time and heavier computation. Therefore, running on a larger output size might allow the effect of core number to become more obvious. Figure 15 demonstrates the test results on a dataset of 9000 transactions with a lower support value, so that the program will produce more and larger outputs. The figure shows a relatively large runtime decrease from 2 cores to 3 cores, though with more than 3 cores, the runtime change seems trivial. For the previous input parameter, there was little computation required for each transaction, and hence the capability of each thread was not largely used up. Thus, if more computation is required for each transaction in the program, the core number tends to have more impact on the runtime.

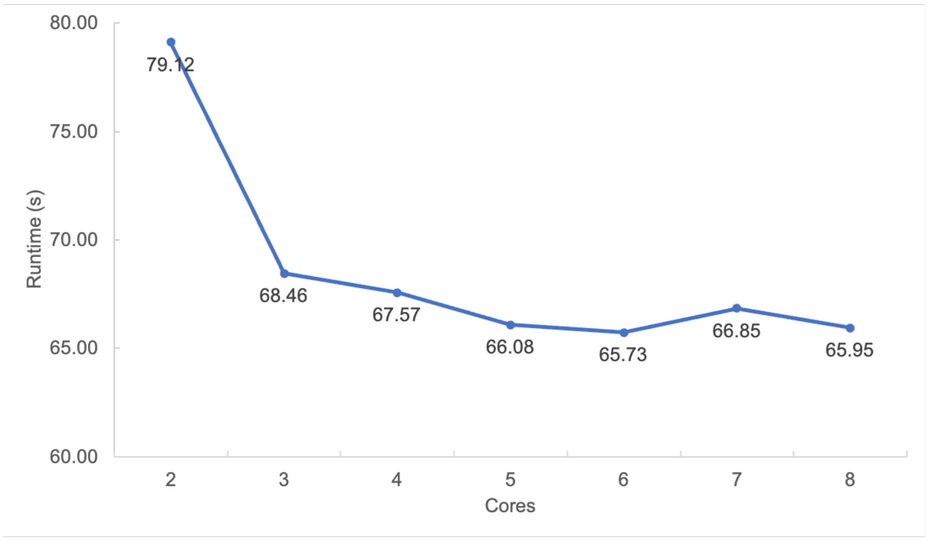


Figure 15: Runtime over core number
(9000 transactions, 0.05% support, 50% confidence)

4 Conclusion

By applying multiple layers of parallelism using different strategies, the resulting parallel Apriori algorithm is able to handle association rule mining more efficiently, no matter what transactional dataset or input parameters are given. With the capabilities provided by Haskell, the parallelism applied to the Apriori algorithm can be programmed in a few lines of code, while producing significant improvement on the program performance.

References

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In *Proc. of 20th intl. conf. on vldb* (pp. 487–499).

Carrie. (2018). *E-commerce data.* Retrieved 2021-12-22, from `https://www.kaggle.com/carrie1/ecommerce-data/home?select=data.csv`

Schiessl, C. (2011). *Implementation of the apriori algorithm in haskell.* Retrieved 2021-12-22, from `https://gist.github.com/cs/2909095`

A Usage

Parallelism on Apriori Algorithm using `parMap` from `Control.Monad.Par`:

```
stack exec apriori-parallel-exe <csv_filename> <min_support> <min_confidence>
```

Execute the sequential implementation:

```
stack exec apriori-parallel-exe <csv_filename> <min_support> <min_confidence> seq
```

Apply parallelism using MapReduce:

```
stack exec apriori-parallel-exe <csv_filename> <min_support> <min_confidence> chunk
```

Specify the size of chunk to reduce to:

```
stack exec apriori-parallel-exe <csv_filename> <min_support> <min_confidence> chunk <chunk_size>
```

B Code Listing

app/Main.hs

```haskell
module Main where
import Apriori ( getAssocRules, getFreqItemsets, getInitFreqItemset,
                aprioriChunk, aprioriByChunkSize,
                getInitFreqItemsetS, getFreqItemsetsS )

import LoadData ( readTableToLst )

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


main :: IO ()
main = do
    args <- getArgs
    case args of
        -- sequential
        [fn, sp, cf, "seq"] -> do
            let filename = fn
                support = read sp :: Double
                confidence = read cf :: Double
            -- get all transactions
            transactions <- readTableToLst filename
            let initFreqItemset = getInitFreqItemsetS support transactions
                freqItemsets = concat $
                    List.unfoldr (getFreqItemsetsS support transactions) initFreqItemset
            print $ getAssocRules confidence transactions freqItemsets
        -- parMap
        [fn, sp, cf] -> do
            let filename = fn
                support = read sp :: Double
                confidence = read cf :: Double
            -- get all transactions
            transactions <- readTableToLst filename
            let initFreqItemset = getInitFreqItemset support transactions
```

```haskell
                freqItemsets = concat $
                    List.unfoldr (getFreqItemsets support transactions) initFreqItemset
            print $ getAssocRules confidence transactions freqItemsets
        -- mapReduce
        [fn, sp, cf, "chunk"] -> do
            let filename = fn
                support = read sp :: Double
                confidence = read cf :: Double
            -- get all transactions
            transactions <- readTableToLst filename
            print $ aprioriChunk transactions support confidence
        -- mapReduce with chunk size
        [fn, sp, cf, "chunk", cs] -> do
            let filename = fn
                support = read sp :: Double
                confidence = read cf :: Double
                chunkSize = read cs :: Int
            -- get all transactions
            transactions <- readTableToLst filename
            print $ aprioriByChunkSize chunkSize transactions support confidence
        _ -> do
            pn <- getProgName
            die $ "Usage: stack exec "++pn
                ++" <csv_filename> <min_support> <min_confidence> [seq] | [chunk] | [chunk <chunk_size>]"
```

src/Apriori.hs [Reference: (Schiessl, 2011)]

```haskell
module Apriori where

import qualified Data.List as List
import qualified Data.Set as Set
import Control.Monad ( guard )
import Control.Monad.Par ( runPar, parMap)
import Control.DeepSeq ( NFData(..) )
import Control.Parallel.Strategies (parList, using, rdeepseq, parBuffer)

newtype Itemset = Itemset (Set.Set String) deriving (Eq, Ord)
instance NFData Itemset where
  rnf (Itemset i) = rnf i

data AssocRule = AssocRule (Set.Set String) (Set.Set String) Double Double deriving (Eq, Ord)
instance Show AssocRule where
    show (AssocRule a b s c) =
        "\n" ++ show a ++ " => " ++ show b ++ "  (" ++ show s ++ ", " ++ show c ++ ")"

instance NFData AssocRule where
    rnf (AssocRule a b s c) = rnf a `seq` rnf b `seq` rnf s `seq` rnf c



getSupport :: [Itemset] -> Itemset -> Double
getSupport transactions (Itemset i) =
    fromIntegral (supportCount i) / fromIntegral (length transactions)
    where supportCount i = length $
            filter (Set.isSubsetOf i) $ map (\(Itemset x) -> x) transactions

getConfidence :: [Itemset] -> Itemset -> Itemset -> Double
getConfidence transactions (Itemset a) (Itemset b) =
    getSupport transactions (Itemset $ a `Set.union` b) / getSupport transactions (Itemset a)

```

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

getAssocRules :: Double -> [Itemset] -> [Itemset] -> [AssocRule]
getAssocRules minConfidence transactions sets = do
    Itemset is <- sets
    subset <- Set.toList $ Set.powerSet is
    let s = is `Set.difference` subset
    guard $ not (Set.null s) && (s /= subset)
    let conf = getConfidence transactions (Itemset subset) (Itemset s)
    guard $ conf > minConfidence
    let supp = getSupport transactions (Itemset subset)
        rule = AssocRule subset s supp conf
    return rule


---- sequential ----
getInitFreqItemsetS :: Double -> [Itemset] -> [Itemset]
getInitFreqItemsetS minSupport transactions =
    let initCandItemset = removeDup $
            concatMap (\(Itemset t) -> map (Itemset . Set.singleton) $ Set.toList t) transactions
    --- sequential
    in filter (\cand -> getSupport transactions cand > minSupport) initCandItemset

aprioriGenS :: [Itemset] -> [Itemset]
aprioriGenS iss =
    let
        -- join step
        selfJoin = [Itemset (a `Set.union` b)
            | (Itemset a) <- iss, (Itemset b) <- iss, validateCandidate a b]
        validateCandidate a b = Set.size (a `Set.difference` b) == 1
        -- prune step
        nonFrequentSubsets (Itemset i) = all (\s -> Itemset s `elem` iss) (properSubsets i)
        powerSetList s = Set.toList $ Set.powerSet s
        properSubsets s = filter (\x -> Set.size x == Set.size s - 1) (powerSetList s)
        --- sequential
        candItemset = filter nonFrequentSubsets selfJoin
    in removeDup candItemset

getFreqItemsetsS :: Double -> [Itemset] -> [Itemset] -> Maybe ([Itemset], [Itemset])
getFreqItemsetsS _ _ [] = Nothing
getFreqItemsetsS minSupport transactions currFreqItemset =
    let nextCandItemset = aprioriGenS currFreqItemset
        --- sequential
        nextFreqItemset = filter (\cand -> getSupport transactions cand > minSupport) nextCandItemset
    in Just (currFreqItemset, nextFreqItemset)


---- parMap ----
getInitFreqItemset :: Double -> [Itemset] -> [Itemset]
getInitFreqItemset minSupport transactions =
    let initCandItemset = removeDup $
            concatMap (\(Itemset t) -> map (Itemset . Set.singleton) $ Set.toList t) transactions
    --- parMap
        initFreqItemset = runPar $
            parMap (\cand -> (cand, getSupport transactions cand > minSupport)) initCandItemset
    in concat $ [[cand] | (cand, isFreq) <- initFreqItemset, isFreq]

aprioriGen :: [Itemset] -> [Itemset]
aprioriGen iss =
    let
        -- join step
        selfJoin = [Itemset (a `Set.union` b)
            | (Itemset a) <- iss, (Itemset b) <- iss, validateCandidate a b]
        validateCandidate a b = Set.size (a `Set.difference` b) == 1
        -- prune step
        nonFrequentSubsets (Itemset i) = all (\s -> Itemset s `elem` iss) (properSubsets i)
        powerSetList s = Set.toList $ Set.powerSet s
        properSubsets s = filter (\x -> Set.size x == Set.size s - 1) (powerSetList s)
        --- parMap
        candItemsetLst = runPar $ parMap (\cand -> (cand, nonFrequentSubsets cand)) selfJoin
        candItemset = concat $ [[cand] | (cand, isSubSet) <- candItemsetLst, isSubSet]
    in removeDup candItemset

getFreqItemsets :: Double -> [Itemset] -> [Itemset] -> Maybe ([Itemset], [Itemset])
getFreqItemsets _ _ [] = Nothing
getFreqItemsets minSupport transactions currFreqItemset =
    let nextCandItemset = aprioriGen currFreqItemset
        --- parMap
        nextFreqItemsetLst = runPar $
            parMap (\cand -> (cand, getSupport transactions cand > minSupport)) nextCandItemset
        nextFreqItemset = concat $ [[cand] | (cand, isFreq) <- nextFreqItemsetLst, isFreq]
    in Just (currFreqItemset, nextFreqItemset)


--- MapReduce ----
apriori :: Double -> Double -> [Itemset] -> [Itemset] -> [AssocRule]
apriori support confidence transactions transAll =
    let initFreqItemset = getInitFreqItemsetC support transactions transAll
        freqItemsets = concat $ List.unfoldr (getFreqItemsets support transAll) initFreqItemset
    in getAssocRules confidence transAll freqItemsets

getInitFreqItemsetC :: Double -> [Itemset] -> [Itemset] -> [Itemset]
getInitFreqItemsetC minSupport transactions transAll =
    let initCandItemset = removeDup $
            concatMap (\(Itemset t) -> map (Itemset . Set.singleton) $ Set.toList t) transactions
    --- parMap
        initFreqItemset = runPar $
            parMap (\cand -> (cand, getSupport transAll cand > minSupport)) initCandItemset
    in concat $ [[cand] | (cand, isFreq) <- initFreqItemset, isFreq]


aprioriByChunkSize :: Int -> [Itemset] -> Double -> Double -> [AssocRule]
aprioriByChunkSize n transactions support confidence =
    removeDup (concatMap (\c -> apriori support confidence c transactions) chunks
            `using` parBuffer 100 rdeepseq)
        where
            chunk _ [] = []
            chunk n xs = let (as,bs) = splitAt n xs in as : chunk n bs
            chunks = chunk n transactions

aprioriChunk :: [Itemset] -> Double -> Double -> [AssocRule]
aprioriChunk transactions support confidence =
    removeDup (concatMap (\c -> apriori support confidence [c] transactions) transactions
            `using` parBuffer 100 rdeepseq)
```

src/LoadData.hs

```haskell
module LoadData where
```

```haskell
import Apriori ( Itemset(..) )
import qualified Data.Set as Set
import Text.CSV (parseCSVFromFile)
import System.Exit (die)


-- read the csv file to get a list of transactions
readTableToLst :: FilePath -> IO [Itemset]
readTableToLst filename = do
    csv_file <- parseCSVFromFile filename
    case csv_file of
        Right csv -> return $ getTransactions csv
        Left err -> die $ show err
    where
        getItemset record = Itemset $ Set.fromList $ filter (/= "") record
        getTransactions csv = map getItemset csv
```