COMS4995 PFP Final Project

Hsuan-Ru Yang (hy2713)

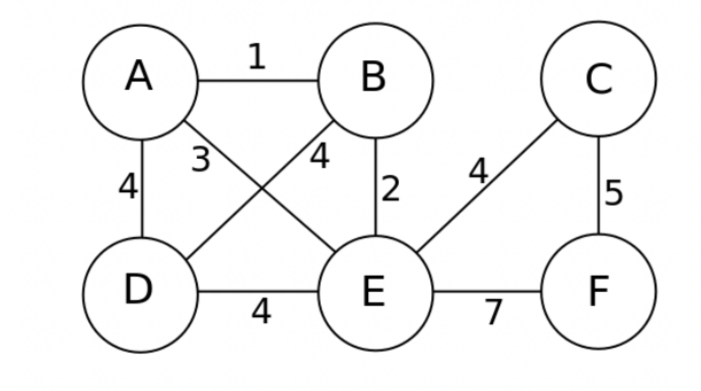
Introduction

In this project I will seek to implement a parallel version of Kruskal’s Algorithm, targeted to solve the "Minimum Spanning Tree" problem. I will focus on a specific variant of the algorithm, namely the filterKruskal algorithm, which is better suited for parallelism.

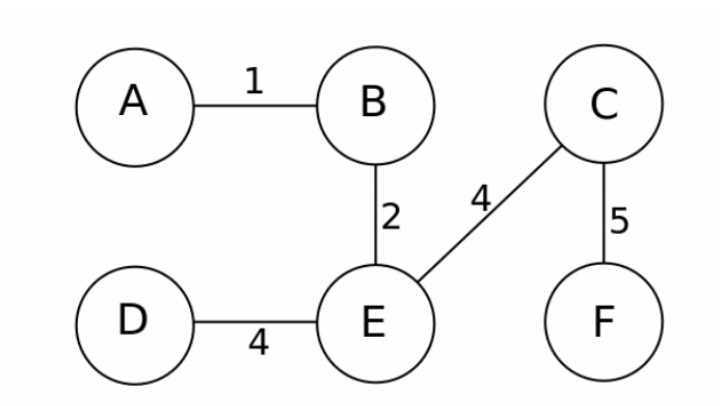
Background

Minimum Spanning Tree

A minimum spanning tree in a connected graph is a subset of the edges of the graph that connects all vertices in the graph, forms no cycle, and has the smallest total edge weight.
For example in the following graph (https://en.wikipedia.org/wiki/Minimum_spanning_tree):



A minimum spanning tree would be:



Kruskal’s Algorithm

One of the algorithms for finding the minimum spanning tree is Kruskal’s algorithm. The basic idea of the algorithm is to greedly add the edge with the smallest weight that connects two vertices from different trees, where each node in the graph is initialized as a tree of itself. The algorithm makes use of the union-find data structure, and can be outlined as:

```
algorithm Kruskal(G) is
    F:= ∅
    for each v ∈ G.V do
        MAKE-SET(v)
    for each (u, v) in G.E ordered by weight(u, v), increasing do
        if FIND-SET(u) ≠ FIND-SET(v) then
            F:= F ∪ {(u, v)} ∪ {(v, u)}
            UNION(FIND-SET(u), FIND-SET(v))
    return F
```

Filter-Kruskal

The filter-Kruskal algorithm is a variant of the original Kruskal’s algorithm that was designed to suit parallelism. The basic idea is to partition the edges and filter out edges that connect vertices in the same tree. The algorithm can be outlined as follows:

```
function filter_kruskal(G) is
    if |G.E| < kruskal_threshold:
        return kruskal(G)
    pivot = choose_random(G.E)
    $E_{\leq}$, $E_{>}$ = partition(G.E, pivot)
    A = filter_kruskal($E_{\leq}$)
    $E_{>}$ = filter($E_{>}$)
    A = A ∪ filter_kruskal($E_{>}$)
    return A

function partition(E, pivot) is
    $E_{\leq}$ = ∅, $E_{>}$ = ∅
    foreach (u, v) in E do
        if weight(u, v) <= pivot then
            $E_{\leq}$ = $E_{\leq}$ ∪ {(u, v)}
        else
            $E_{>}$ = $E_{>}$ ∪ {(u, v)}
    return $E_{\leq}$, $E_{>}$

function filter(E) is
    $E_{\text{filtered}}$ = ∅
    foreach (u, v) in E do
        if find_set(u) ≠ find_set(v) then
            $E_{\text{filtered}}$ = $E_{\text{filtered}}$ ∪ {(u, v)}
    return $E_{\text{filtered}}$
```

The algorithm is easier to parallelize since sorting, filtering, and partitioning can be implemented in a parallel manner by distributing the edges among the processors.

Implementation

Union-Find

A union-find data structure should support two main operations: the find-root operation that finds the representation of a given node, and a merge operation that merges two sets. Here we use the implementation from https://github.com/jgrant27/jngmisc/blob/master/haskell/quick_union.hs

```haskell
import Data.Sequence as Seq
data DisjointSet = DisjointSet
     { count :: Int, ids :: (Seq Int), sizes :: (Seq Int) }
     deriving (Read,  Show)
-- Return id of root object
findRoot :: DisjointSet -> Int -> Int
findRoot set p | p == parent = p
               | otherwise   = findRoot set parent
               where
                parent = index (ids set) (p - 1)
-- Are objects P and Q connected ?
connected :: DisjointSet -> Int -> Int -> Bool
connected set p q = findRoot set q `par` (findRoot set p) == (findRoot set q)
-- Replace sets containing P and Q with their union
quickUnion :: DisjointSet -> Int -> Int -> DisjointSet
quickUnion set p q | i == j = set
                   | otherwise = DisjointSet cnt rids rsizes
                     where
                        (i, j)   = findRoot set q `par` (findRoot set p, findRoot set q)
                        (i1, j1) = (index (sizes set) (i - 1), index (sizes set) (j - 1))
                        (cnt, psmaller, size) = (count set - 1, i1 < j1, i1 + j1)
                        -- Always make smaller root point to the larger one
                        (rids, rsizes) = if psmaller
                                             then (update (i - 1) j (ids set), update (j - 1)
size (sizes set))
                                             else (update (j - 1) i (ids set), update (i - 1)
size (sizes set))
```

Kruskal's Algorithm

Here we provide the implementation of the original Kruskal's algorithm:

```
kruskal  :: [(Int, Int, Int)] -> [(Int, Int, Int)] -> DisjointSet -> ([(Int, Int, Int)],
DisjointSet)
kruskal [] t m = (t, m)
kruskal e t m
  | diff_edge m (u, v, w) = kruskal (tail e) ((u,v,w):t) (quickUnion m (u+1) (v+1))
  | otherwise = kruskal (tail e) t m
  where
    (u, v, w) = head e
```

Filter-Kruskal

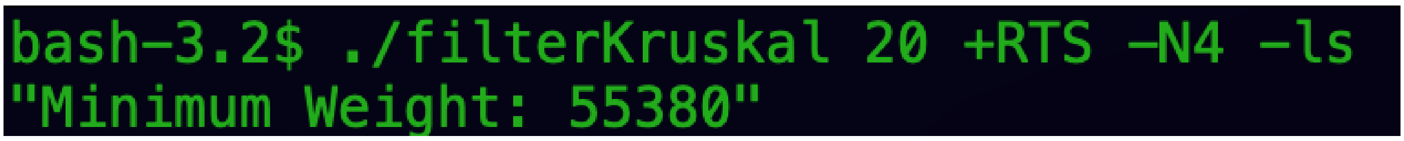
The following is my implementation of the filter-Kruskal's algorithm, which includes the key components *filter* and *partition*.

```
filter' :: [(Int, Int, Int)] -> DisjointSet -> [(Int, Int, Int)]
filter' e m = List.filter (diff_edge m) e

partitionEdges :: [(Int, Int, Int)] -> Int -> ([(Int, Int, Int)], [(Int, Int, Int)])
partitionEdges e pivot = List.partition (compare_pivot pivot) e

filterKruskal :: [(Int, Int, Int)] -> DisjointSet -> Int -> Int -> Int -> Int -> ([(Int, Int,
Int)], DisjointSet)
filterKruskal e m depth maxDepth threshold sortDepth
  | (depth > maxDepth) || (List.length e < threshold) = kruskal (sortEdges sortDepth e) [] m
  | otherwise =  (res_l ++ res_r, new_m_r)
    where
      (_, _, pivot) = head e
      (e_l, e_r) = partitionEdges e pivot
      (res_l, new_m_l) = filterKruskal e_l m (depth+1) maxDepth threshold sortDepth
      e_r' = filter' e_r new_m_l
      (res_r, new_m_r) = filterKruskal e_r' new_m_l (depth+1) maxDepth threshold sortDepth
```

A sample output of the program would be:



```
bash-3.2$ ./filterKruskal 20 +RTS -N4 -ls
"Minimum Weight: 55380"
```

Parallelization

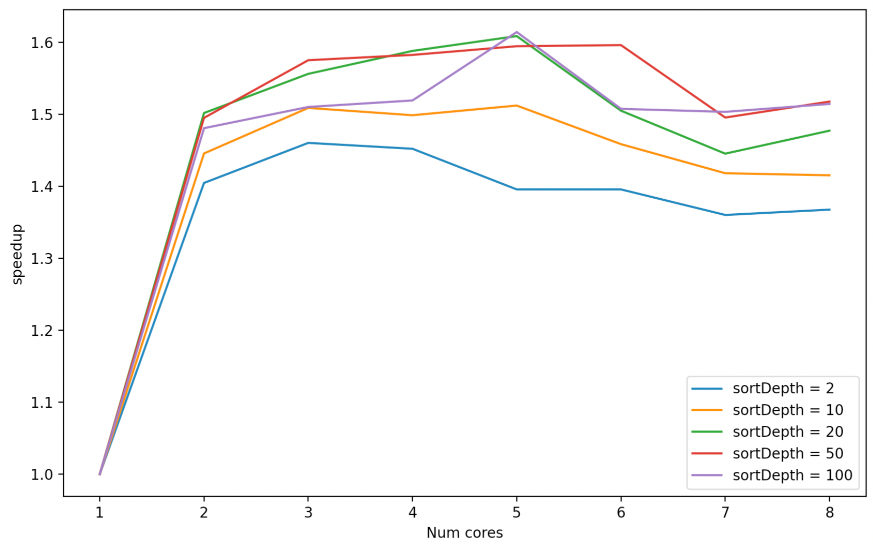
This project focuses on parallelizing three parts of the algorithm: 1. The sorting part for the base case original Kruskal's algorithm, which requires the input edges to be in ascending order. 2. The filtering step and 3. The partition step in the filter-Kruskal's algorithm. In the following section I will discuss the parallelized implementation of these three components. All experiments in this section were run on a graph with 3000 nodes and 100000 edges, and the reported numbers are the average of 4 runs of each program.

Sorting

The parallelized version of sorting is implemented as the following:

```haskell
sortEdges :: Int -> [(Int, Int, Int)] -> [(Int, Int, Int)]
sortEdges n (x:xs)
  | n > 0 = b `par` a ++ x:b
  | otherwise = a ++ x:b
  where
    a = sortEdges (n-1) $ parFilter (\h -> compareEdges h x /= GT) xs
    b = sortEdges (n-1) $ parFilter (\h -> compareEdges h x == GT) xs
sortEdges _ [] = []
```

This is a modified version of the quicksort algorithm, where the first element of the list is chosen as the pivot instead of a random one. The parallel part here is to evaluate the partitioned sub-lists, either greater or less than the pivot, in parallel before it reaches a certain recursion depth. By replacing the sorting function in the original program with this parallelized sorting function, we can get the following figure:



As shown in the figure, the parallelized version of the sorting function indeed provides some speedup when the program is executed on multiple cores.

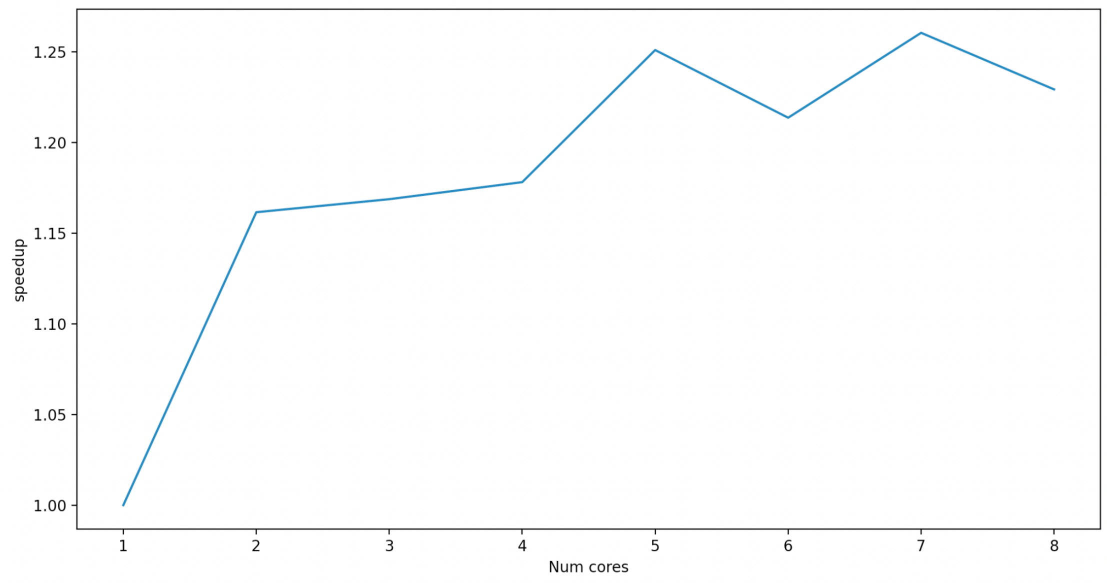
Filter

For the filtering component in the algorithm, the parallelized version is implemented as the following:

```haskell
parFilter :: (a -> Bool) -> [a] -> [a]
parFilter p = withStrategy (parList rseq) . List.filter p

filter' :: [(Int, Int, Int)] -> DisjointSet -> [(Int, Int, Int)]
filter' e m = parFilter (diff_edge m) e
```

Where the filtering function is applied with the *parList* strategy to each element of the list to be filtered. Replacing the original filtering function with this, we can get speedup results shown below:



As shown in the figure above, the program was able to gain some speedup with more cores. However, the speedup is not very significant, probably because filtering is not a very computationally extensive task in the first place, so the benefits of parallelizing it are not obvious.

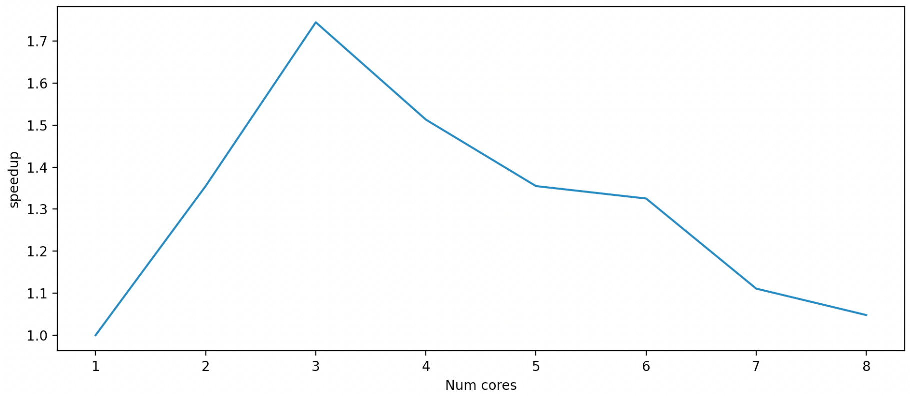
Partition

The parallelized version of partitioning is implemented as below:

```haskell
partitionEdges :: [(Int, Int, Int)] -> Int -> ([(Int, Int, Int)], [(Int, Int, Int)])
partitionEdges [] _ = ([], [])
partitionEdges e pivot = r `par` (l, r)
  where
    l = parFilter (compare_pivot pivot) e
    r = parFilter (not . (compare_pivot pivot)) e
```

There are two things I did for parallelization here:

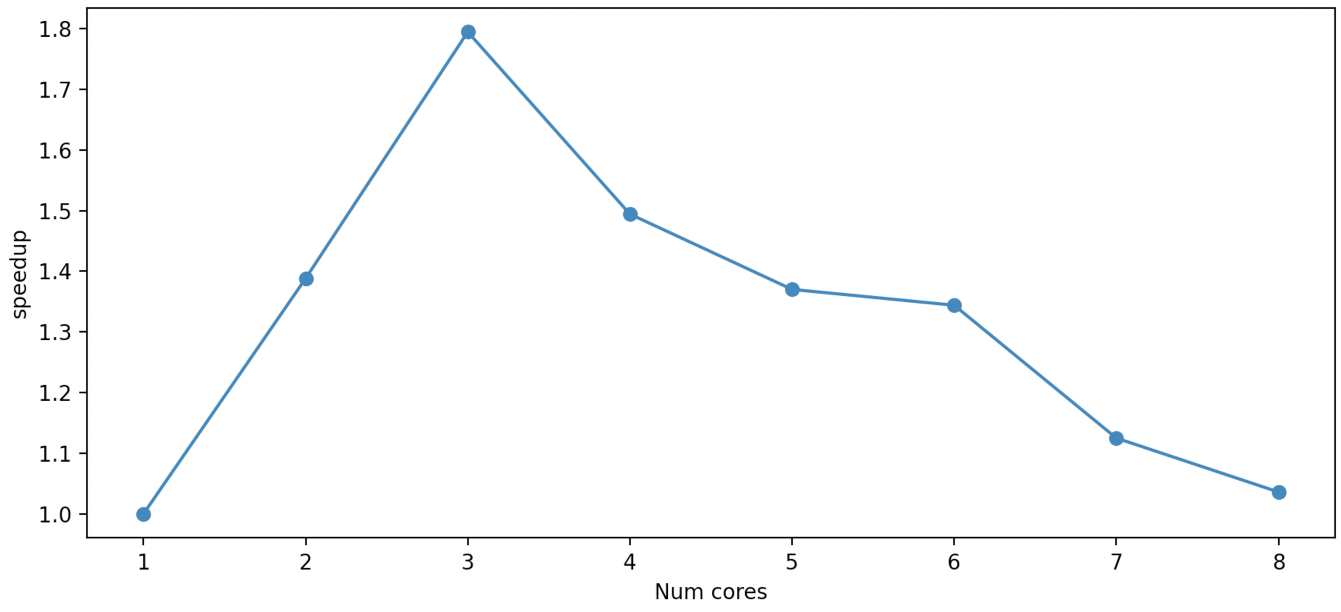
1. Since the goal of the function is to produce two sublists of the input list, with one containing all elements that satisfies a predicament and the other one that contains all elements that doesn't, the filtering of the two sublists are run in parallel with `` r `par` (l, r) ``
2. I also leveraged the `parFilter` function from the previous section to further improve parallelism.
After incorporating these changes, the speedup of the program is shown below:



As shown in the figure, the program is having pretty nice speedup gains in the beginning. However, as the number of cores increases, the speedup ratio starts to decline. This might be due to the fact that partitioning or filtering does not require a lot of computation in the first place, causing the benefits of parallelism being overshadowed by the parallel overheads.

Putting it Together

By putting all three parallelized components together in the original program, we can get the following figure:



We can see in the figure that the speedup gains with 2 or 3 cores is somewhat decent (1.8x speedup with 3 cores). However, speedup decreases when the number of cores keeps increasing. We can also take a look at the number of sparks created:

N2		
converted	193821	9.37%
overflowed	177171	8.57%
dud	0	0.00%
GC	1085371	52.49%
fizzled	300239	14.52%
total	2067898	100.00%

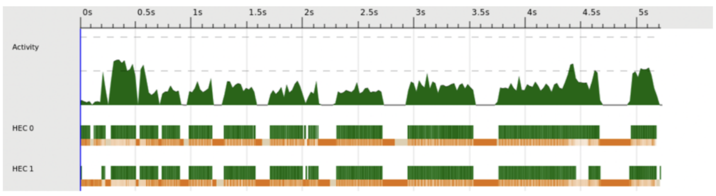
N6		
converted	539942	25.88%
overflowed	96863	4.64%
dud	0	0.00%
GC	628943	30.15%
fizzled	664831	31.87%
total	2086227	100.00%

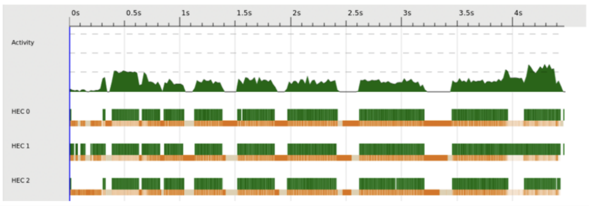
N3		
converted	419898	20.32%
overflowed	110452	5.35%
dud	0	0.00%
GC	961995	46.56%
fizzled	426197	20.63%
total	2065998	100.00%

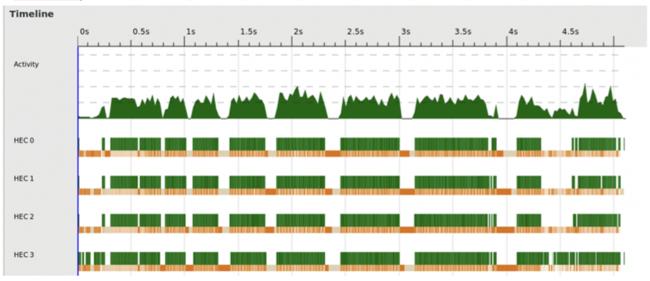
N8		
converted	718080	34.91%
overflowed	104794	5.09%
dud	0	0.00%
GC	520462	25.30%
fizzled	623402	30.31%
total	2056850	100.00%

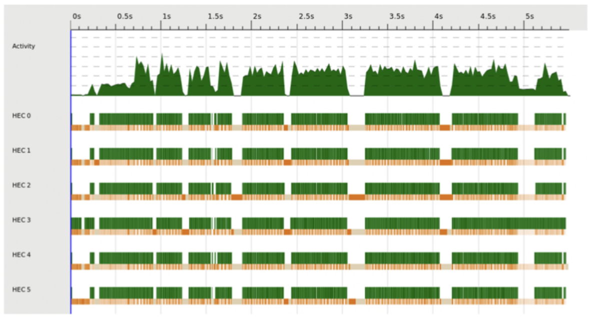
N4		
converted	399628	19.29%
overflowed	90287	4.36%
dud	0	0.00%
GC	848253	40.94%
fizzled	512408	24.73%
total	2071760	100.00%

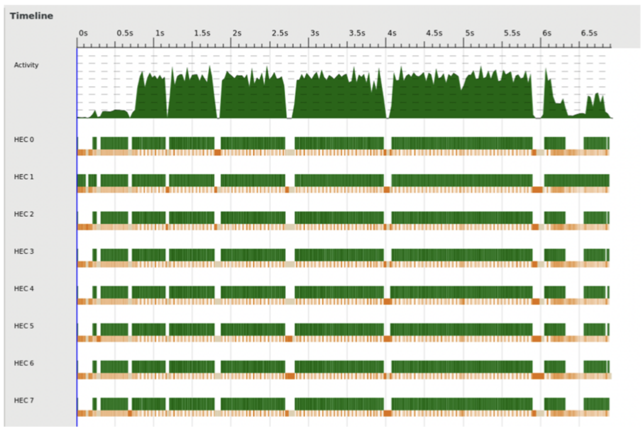
As shown in the tables, the total number of sparks created are quite consistent regardless of the number of cores used, while the number of converted ones increases with more cores. This does not align with the pattern with the speedup chart, however. Here we also present the threadscope graphs for further analysis:











We can see that all the cores are actually pretty busy even with 6 or 8 cores running. Nonetheless, the program still takes longer to execute, compared to 3 or 4 cores. This suggests that the parallel overhead in the program has greatly dominated over the benefits of running the program parallelly. Another possible reason that the speedup ratio is far lower than the theoretical ratio (N times speedup with N cores) is that the whole algorithm depends on a shared union-find data structure. This means that in many steps of the algorithm, the step has to wait until the union-find data structure is finished updating by the previous step in order to proceed. This inevitably leads to a huge part of the program being executed sequentially and not possible to parallelize, which further limits the potential speedup gain of the program, according to Amdalh's Law.

Conclusion

In this project I implemented a parallel version of the filter-Kruskal's algorithm, and demonstrated speedup on the performance of the program compared to the sequential version. However, some key characteristics of the algorithm, including filtering and partition being not very heavy computational tasks and forced sequential execution due to the dependence of a single shared data-structure, has caused the program to underperform compared to the theoretical maximum speedup.

Code Listing

```
--
-----------------------------------------------------------------------------------------------
-
import Data.List as List
import System.IO
import Control.Parallel(par)
import Control.Parallel.Strategies(withStrategy, parList, rseq)
import System.Environment


--
-----------------------------------------------------------------------------------------------
-
-- Union Find Implementation
-- https://github.com/jgrant27/jngmisc/blob/master/haskell/quick_union.hs
import Data.Sequence as Seq
data DisjointSet = DisjointSet
     { count :: Int, ids :: (Seq Int), sizes :: (Seq Int) }
     deriving (Read,  Show)
-- Return id of root object
findRoot :: DisjointSet -> Int -> Int
findRoot set p | p == parent = p
               | otherwise   = findRoot set parent
               where
                parent = index (ids set) (p - 1)
-- Are objects P and Q connected ?
connected :: DisjointSet -> Int -> Int -> Bool
connected set p q = findRoot set q `par` (findRoot set p) == (findRoot set q)
-- Replace sets containing P and Q with their union
quickUnion :: DisjointSet -> Int -> Int -> DisjointSet
quickUnion set p q | i == j = set
                   | otherwise = DisjointSet cnt rids rsizes
                     where
                        (i, j)   = findRoot set q `par` (findRoot set p, findRoot set q)
                        (i1, j1) = (index (sizes set) (i - 1), index (sizes set) (j - 1))
                        (cnt, psmaller, size) = (count set - 1, i1 < j1, i1 + j1)
                        -- Always make smaller root point to the larger one
                        (rids, rsizes) = if psmaller
                                         then (update (i - 1) j (ids set), update (j - 1)
size (sizes set))
                                         else (update (j - 1) i (ids set), update (i - 1)
size (sizes set))
--
-----------------------------------------------------------------------------------------------
-
```

```haskell
--
---------------------------------------------------------------------------------------------
-
getEdge :: Handle -> IO (Int, Int, Int)
getEdge fileHandle =
      do
            line <- hGetLine fileHandle
            let [u, v, w] = words line
            return ((read u :: Int), (read v :: Int), (read w :: Int))


readGraphFile :: String -> (Handle -> IO a) -> IO [a]
readGraphFile fileName handleFunction = do
      fileHandle <- openFile fileName ReadMode
      res <- readLine fileHandle handleFunction []
      return res

readLine :: Handle -> (Handle -> IO a) -> [a] -> IO [a]
readLine fileHandle handleFunction cur = do
      end <- hIsEOF fileHandle
      if end
        then return cur
        else do
            p <- handleFunction fileHandle
            let cur' = p:cur
            readLine fileHandle handleFunction cur'


readGraph :: IO [(Int, Int, Int)]
readGraph = readGraphFile "large_graph.txt" getEdge


--
---------------------------------------------------------------------------------------------
-

diff_edge :: DisjointSet -> (Int, Int, Int) -> Bool
diff_edge m (u, v, _) = not $ connected m (u+1) (v+1)

compareEdges :: (Int, Int, Int) -> (Int, Int, Int) -> Ordering
compareEdges (u1, v1, w1) (u2, v2, w2) = compare (w1, u1, v1) (w2, u2, v2)

compare_pivot :: Int -> (Int, Int, Int) -> Bool
compare_pivot p (_, _, w) = (w <= p)


--
---------------------------------------------------------------------------------------------
-----------
-- Parallel Stuff
```

```haskell
--
---------------------------------------------------------------------------------------------
-----------


sortEdges :: Int -> [(Int, Int, Int)] -> [(Int, Int, Int)]
sortEdges n (x:xs)
  | n > 0 = b `par` a ++ x:b
  | otherwise = a ++ x:b
  where
    a = sortEdges (n-1) $ parFilter (\h -> compareEdges h x /= GT) xs
    b = sortEdges (n-1) $ parFilter (\h -> compareEdges h x == GT) xs
sortEdges _ [] = []


parFilter :: (a -> Bool) -> [a] -> [a]
parFilter p = withStrategy (parList rseq) . List.filter p


filter' :: [(Int, Int, Int)] -> DisjointSet -> [(Int, Int, Int)]
filter' e m = parFilter (diff_edge m) e

partitionEdges :: [(Int, Int, Int)] -> Int -> ([(Int, Int, Int)], [(Int, Int, Int)])
partitionEdges e pivot = r `par` (l, r)
  where
    l = parFilter (compare_pivot pivot) e
    r = parFilter (not . (compare_pivot pivot)) e




--
---------------------------------------------------------------------------------------------
-----------
-- Core Algorithm
--
---------------------------------------------------------------------------------------------
-----------

kruskal :: [(Int, Int, Int)] -> [(Int, Int, Int)] -> DisjointSet -> ([(Int, Int, Int)],
DisjointSet)
kruskal [] t m = (t, m)
kruskal e t m
  | diff_edge m (u, v, w) = kruskal (tail e) ((u,v,w):t) (quickUnion m (u+1) (v+1))
  | otherwise = kruskal (tail e) t m
  where
    (u, v, w) = head e


filterKruskal :: [(Int, Int, Int)] -> DisjointSet -> Int -> Int -> Int -> Int -> ([(Int, Int,
Int)], DisjointSet)
filterKruskal e m depth maxDepth threshold sortDepth
```

```haskell
  | (depth > maxDepth) || (List.length e < threshold) = kruskal (sortEdges sortDepth e) [] m
  | otherwise =  (res_l ++ res_r, new_m_r)  -- `S.using` (S.parTuple2 S.rseq S.rseq)
    where
      (_, _, pivot) = head e
      (e_l, e_r) = partitionEdges e pivot
      (res_l, new_m_l) = filterKruskal e_l m (depth+1) maxDepth threshold sortDepth
      e_r' = filter' e_r new_m_l
      (res_r, new_m_r) = filterKruskal e_r' new_m_l (depth+1) maxDepth threshold sortDepth



main :: IO()
main = do
  -- let edges = [(0, 1, 10), (0, 2, 6),(0, 3, 5),(1, 3, 15), (2, 3, 4)]
  -- let edges =
[(0,1,7),(1,2,8),(0,3,5),(1,3,9),(1,4,7),(2,4,5),(3,4,15),(3,5,6),(4,5,8),(4,6,9),(5,6,11)]
  edges <- readGraph
  args <- getArgs
  let [sortDepth] = (map (\x -> read x:: Int) args)
  let maxDepth = 100
  let threshold = 1000
  let n = maximum [b+1 | (_,b,_) <- edges]
  let uf = DisjointSet n (Seq.fromList [1..n]) (Seq.replicate n 1)
  let (res, _) = filterKruskal edges uf 0 maxDepth threshold sortDepth
  print ("Minimum Weight: " ++ (show (sum [c | (_,_,c) <- res])))
```