Parallel Functional Programming
Columbia University in the City of New York

Parallel Graph Coloring in Haskell

TEAM

Haneen Abdulrashid Mohammed	ham2156
Ishan Guru	ig2333

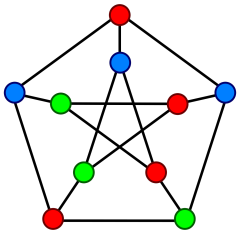
Introduction

Problem

The problem we set out to solve was the graph colouring problem:

Given a graph with n vertices, find a way to assign a colour, given x colours, to each of the vertices such that no two adjacent vertices are of the same colour.

For example, the graph below is a valid, 3-colour solution to our problem:



We have provided both single-threaded and multithreaded solutions to this problem, and done an in-depth comparison of the performances between the two, along with a few other comparisons for problems we encountered along the way.

Approach

We have explored different algorithms to solve graph coloring problem and their potential for parallelization.

1. The first uses backtracking. It sequentially try to assign a color, if it's a valid coloring, to each vertex. If it succeed in assigning a color, we continue to the next node of the graph. If not, we step backtrack and try to find the next available coloring [6]. This approach takes in a fixed number of colours as an input.:

```
G = read graph file
Vertex = get a vertex from G
Backtracking(G, colors_list, Vertex)

Where Backtracking is a recursive function:
For color in colors_list:
        If color is validColor for Vertex:
                Set color of Vertex to color
                If Backtracking(G, colors_list, Vertex+1) == True:
```

Return True
Otherwise unset color of Vertex

2. The second method uses a simple greedy algorithm where for each vertex it finds the lowest allowed color for that vertex [5]:

```
G = read graph file
For each vertex in G do
        Assign smallest allowed color to vertex
End-For
```

3. The next method uses a greedy approach as well that is based on finding independent set. Since vertices in an independent set can be assigned the same color, the algorithm works by repeatedly finding an independent set; assigning color to them; then removes them from the graph [1]:

```
G = read graph file
U =  get all vertices of G
G' = G
While G` is not empty do
        I = get independent set of G'
        Color vertices in I
        U = U - I
        G' = graph induced by U
End-while
```

   where the induced subgraph includes the vertices in U and all the edges that connect vertices in U from the original edges in G [7]

4. The last solution uses a divide-and-conquer approach, where the graph is partitioned, then coloured, then merged. For the divide-and-conquer approach, although we continued to colour using a greedy approach, we removed the restraint on the number of colours that the algorithm can use so that conflicts during merge can easily be resolved. The pseudocode below illustrates the approach we are taking:

```
G= read graph file
partition the graph
for each partition, in parallel:
        colour the partition
for partitions, in parallel:
        merge the partitions while resolving conflict
```

Our goal for parallelization was to find a way to parallelize the single-threaded solutions and do a comparison in time taken to colour the graph. For backtracking and greedy algorithms we found little

opportunity for parallelization; instead we opted to parallelize solving multiple graphs at the same time. For the last approach, the technique we followed was to colour each individual subgraph in parallel and then merge.

The output of our both solutions is True, along with the colouring, if it is valid, or False if we're unable to find one. Our hypothesis is that in both scenarios, time taken to find a valid solution would decrease; however, in the second scenario, where we are splitting the graph into multiple subgraphs, we would need an increased number of colours to find a valid solution.

Testing

In order to properly test our solution, there were a variety of methods needed, such as checking whether or not a graph is a valid colouring. In the sequential algorithm, none of these methods are parallelized, however, in the parallelized version, these methods have also been parallelized as they also contribute to performance. Each of these methods have been included in the appendix section of this paper.

Our test data was acquired from the following sources:

1. LinkedIn Engineering [2]
2. Manually constructed graphs in our format by hand

Once we had decided on a graph representation to use, scripts were written in order to transform graphs from the sources above to be used with our algorithm. This let us ensure that our sequential solution and test methods were correct to begin with, as we could compare with graphs we knew had a solution. Moreover, it let us test on graphs that had a variety of complexity, which let us better ensure our colourings were valid. Links to these data sets have been provided in the appendix.

Running the Code

We've structured our code in a way that lets you choose which algorithm to run, along with whether you're operating on a file or a folder. The examples below show you how to run each of our algorithms. To run any algorithm in parallelized form, simply add "+RTS -NX" as runtime parameters, where X is the number of threads to run on.

The sequence of commands below are a walk through of how to run each of our algorithms

```
-- install dependencies
./install.sh


-- make project
make


-- Program usage
Usage: graph_colouring <graph-{file/folder}name/> <number-of-colors> <algo:
{divide-conquer/backtracking/indep-set/greedy}> <method: file/folder>
<output-folder>


-- greedy sequential approach
./graph_colouring samples/CLIQUE_300.3color 3 greedy file results


-- divide-and-conquer sequential approach
./graph_colouring samples/CLIQUE_300.3color 300 divide-conquer file results


-- divide-and-conquer parallel approach
./graph_colouring samples/CLIQUE_300.3color 300 divide-conquer file results
+RTS -N<X>
```

```
-- independent set sequential approach

./graph_colouring samples/CLIQUE_300.3color 300 indep-set file results


-- independent set parallel approach

./graph_colouring samples/CLIQUE_300.3color 300 indep-set file results +RTS

-N<X>

-- solve folder of graphs using greedy approach

./graph_colouring test/ 4 greedy folder results +RTS -N1 -ls
```

Results

In order to come up with a detailed comparison, we tested with the following variables against both our algorithms:

- Number of nodes in graph
- Number of threads (1 thread in the sequential solution, N threads in the parallelized solution)

Additionally, we split testing into two categories - the first being testing individually on graphs with a set number of nodes (300, 1800, 3000) and the second on a complete directory of graphs passed as an input parameter. Testing on the directory involved sequentially solving each graph in the sequential solution and outputting the result to an output directory; and solving each graph in parallel in the parallelized solution, and outputting the result to an output directory. In this scenario, we tested on graphs we knew had a solution in order to be able to ensure that a solution was found for each graph.

For the parallel scenario where graphs are split into subgraphs, we gave a very high upper bound of colours available to use for conflict resolution. Since we were using a greedy backtracking approach, we knew the algorithm would only use colours as needed. The output files generated for each test contain the number of colours used to find a solution. Once again, this testing was conducted on graphs of fixed node sizes of 300, 1800, and 3000.

Since we were unable to parallelize the purely greedy algorithm, we decided to test it against a folder of graphs instead of individual graphs. We used a folder of 50 graphs with a variety of node counts and complexity (Source: LinkedIn Engineering [2]) and first computed time to solve sequentially, and then in parallel (split by graphs to solve).

Testing was conducted on a Mid 2019 Macbook Pro with a 2.6Ghz 6-Core Intel i7 and 16GB of RAM using the following files:

300 nodes: CLIQUE_300_3.3color
1800 nodes: BENCH_1800.3color
3000 nodes: RAND_3000_3_80.3color

Sequential Solution Benchmarks

50-graph input folder

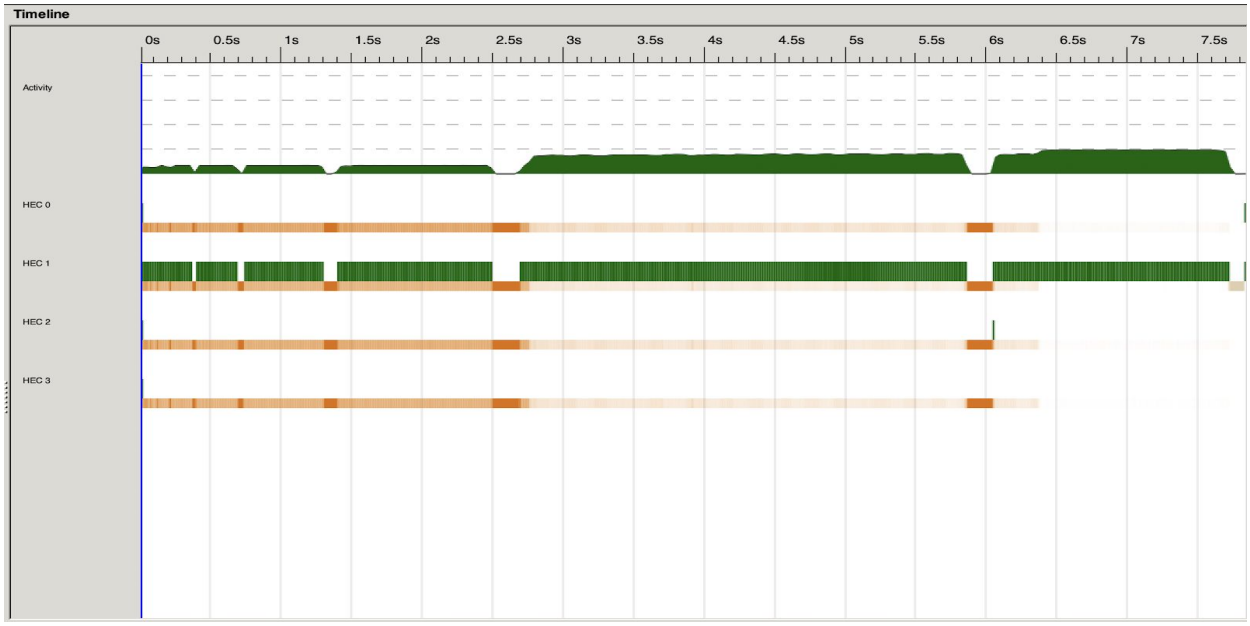
Sequentially solving the graph input folder using our greedy-backtracking algorithm yielded the following results:

Number of Files	Time Taken (s)

50	139.183

Greedy-Backtracking Approach

The following results show the performance of our sequential solution across graphs with 300, 1800, and 3000 nodes. As seen below, in each case, only one thread is used in the execution of this task.

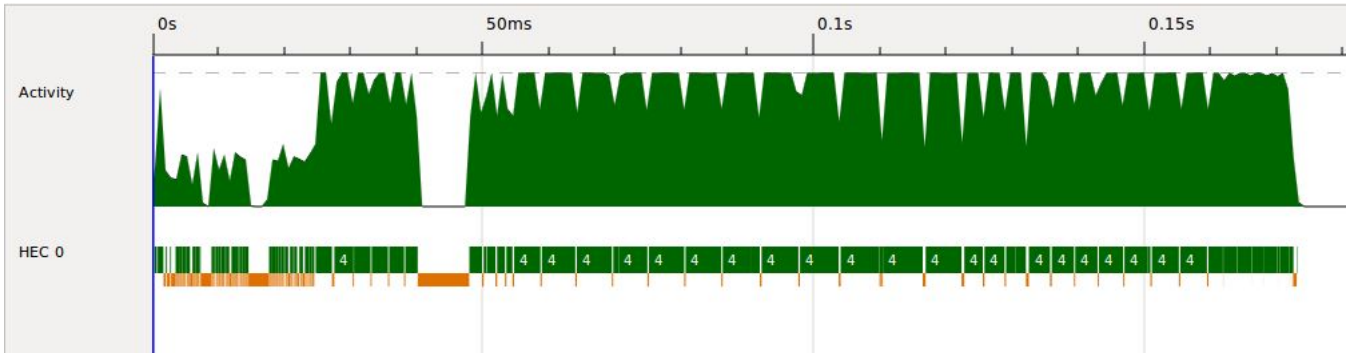


Number of Nodes	Time Taken (s)
300	0.080
1800	1.435
3000	8.166

Independent Set Approach

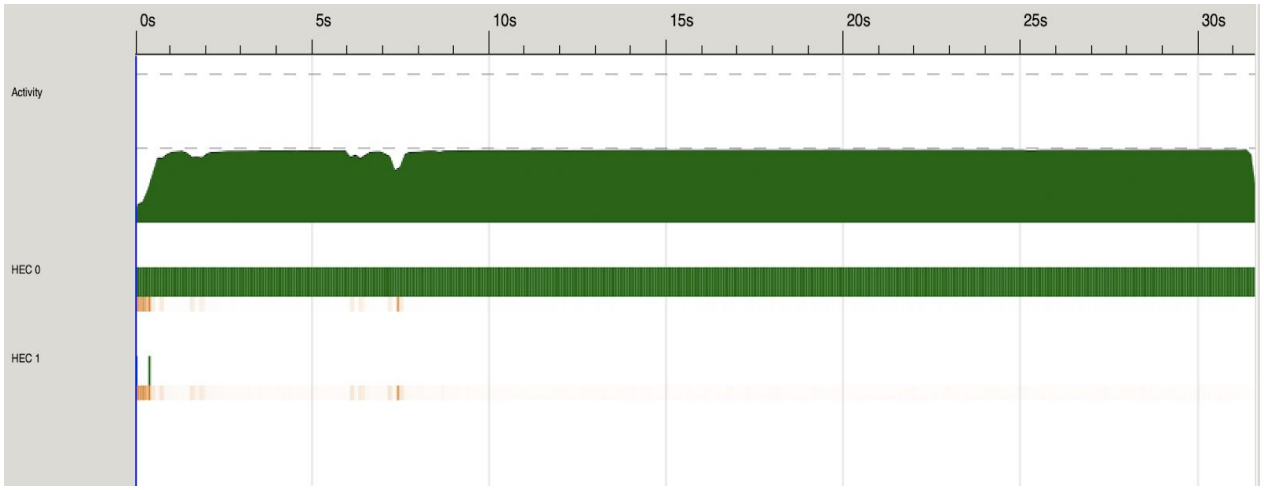
The following results show the performance of our sequential solution across graphs with 300, 1800, and 3000 nodes. As seen below, in each case, only one thread is used in the execution of this task.

Number of Nodes	Time Taken (s)
300	0.181
1800	10.84
3000	112.811



Divide-and-Conquer Approach

The following results show the performance of our sequential solution across graphs with 300, 1800, and 3000 nodes. As seen below, in each case, only one thread is used in the execution of this task.



Number of Nodes	Time Taken (s)
300	0.185
1800	28.078
3000	148.924

As seen from the results above, our divide-and-conquer approach is much slower than the greedy approach, however, we believe that this approach will have a better performance increase from parallelization than the greedy approach. We believe this approach is much slower because of the additional merge computation that's required when merging the two coloured subgraphs.

Parallelized Solution Benchmarks

In order to effectively determine what to parallelize we studied chunks of our sequential approach and noticed that there were three main candidates

1. Reading the graph
2. Finding a solution to the graph
3. Determining whether the graph is valid

As previously mentioned, the parallelization techniques for each of our algorithms were as follows:

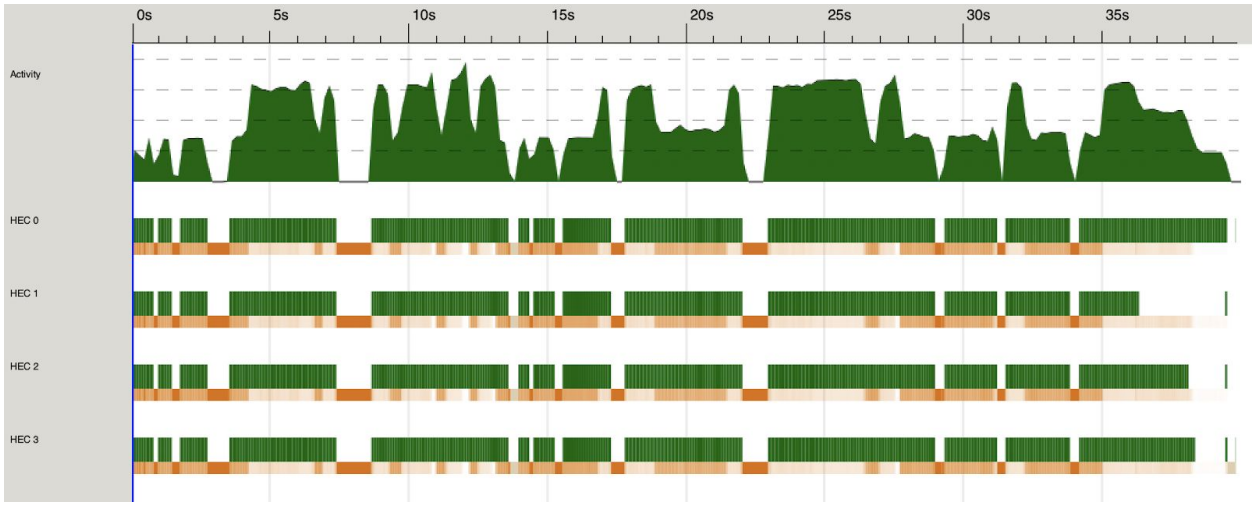
1. Assign a colour in parallel and search with that input - for the greedy-backtracking approach
2. Colour each partition of the graph in parallel - for the divide-and-conquer approach. The merge step also involves conflict resolution, which ensures post-merge that a solution is still valid.

50-graph input folder

As previously mentioned, we were unable to parallelize the greedy-backtracking algorithm itself. We decided to parallelize IO and use the algorithm to solve a folder of graphs instead of singular graphs. Solving the graphs in parallel in the input folder using our greedy-backtracking algorithm yielded the following results:

Number of Files	Number of Cores	Time Taken (s)	Speedup
50	1	130.803	1.0
50	2	50.888	2.57
50	3	40.136	3.25
50	4	36.138	3.62
50	5	40.104	3.26
50	6	50.129	2.61

Compared to our sequential approach, the parallel approach gives us a 3.62x best case performance increase, but also effectively distributes work over the number of threads. As seen below, this parallelization is also quite evenly distributed over the number of threads we are operating on.



Parallel Independent Set Approach

The independent set graph coloring algorithm has four main routines:

1. Find and independent set for a vertex in the graph -> I
2. Update the list of vertices by removing from it the set of vertices in I -> U
3. Color the set of vertices in I
4. Create an induced subgraph from U

Only 2, 3 and 4 can be parallelized, since all the subsequent steps depends on step 1.
To parallelize step 4, we used map with parListChunk, to reduce the overhead of running map in parallel. From the results below, we can see there is no speed up gained by increasing the number of cores used. We hypothesize that the reason is that the first step is the most time consuming part which can't be parallelized.

Number of Cores	Number of Nodes	Time Taken (s)	Speedup
1	300	0.181	1.0
2	300	0.181	1.0
3	300	0.181	1.0
4	300	0.181	1.0
5	300	0.181	1.0
6	300	0.181	1.0

Number of Cores	Number of Nodes	Time Taken (s)	Speedup
1	1800	10.84	1.0
2	1800	10.691	1.01
3	1800	10.621	1.02
4	1800	10.562	1.03
5	1800	10.741	1.01
6	1800	10.691	1.01

Number of Cores	Number of Nodes	Time Taken (s)	Speedup
1	3000	112.811	1.0
2	3000	112.501	1.002
3	3000	111.471	1.012
4	3000	111.141	1.015
5	3000	108.331	1.04
6	3000	108.591	1.04

Parallel Divide-and-Conquer Approach

As previously mentioned, the divide and conquer approach works as follows:

1. Break the graph into subgraphs
2. Colour each part individually (in parallel)
3. Merge the subgraphs (should be done in parallel, need to optimize this)

For our parallel strategy, we decided to use static partitioning in the divide and conquer approach to ensure we are able to write and test a merge function that resolves conflicts as expected. With dynamic partitioning, we found that our merge function was not able to effectively resolve conflicts in colours, leading to an incorrect colouring. Our assumption is that a x2 partitioning should get us almost double the performance. *Note: A minimal number colouring is not possible in this solution (which was three on the graphs we used for testing this) because of the conflict resolution that is needed during subgraph merge.*

Number of Threads	Number of Nodes	Time Taken (s)	Speedup
1	300	0.201	1.00
2	300	0.169	1.12
3	300	0.174	1.15
4	300	0.143	1.41
5	300	0.158	1.27
6	300	0.138	1.46

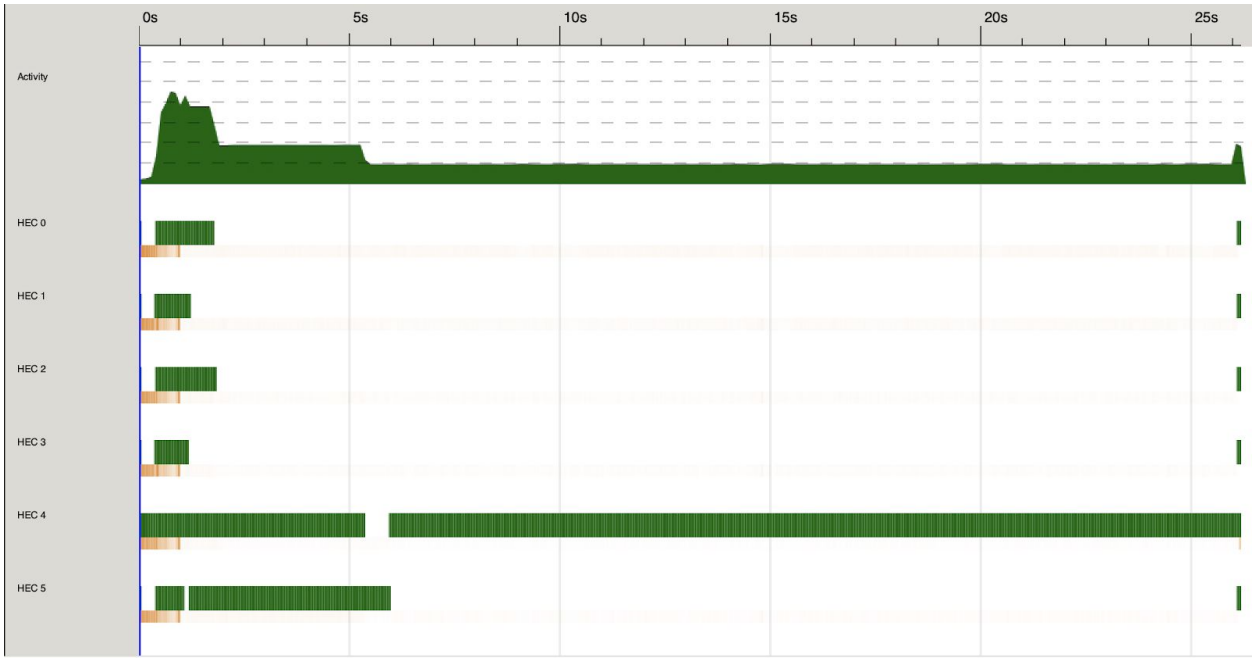
Number of Threads	Number of Nodes	Time Taken (s)	Speedup
1	1800	29.245	1.00
2	1800	27.400	1.07
3	1800	27.370	1.07
4	1800	26.835	1.09
5	1800	27.788	1.05
6	1800	26.215	1.12

Number of Threads	Number of Nodes	Time Taken (s)	Speedup
1	3000	153.365	1.00

2	3000	139.396	1.10
3	3000	133.767	1.10
4	3000	136.388	1.15
5	3000	110.431	1.39
6	3000	109.273	1.40

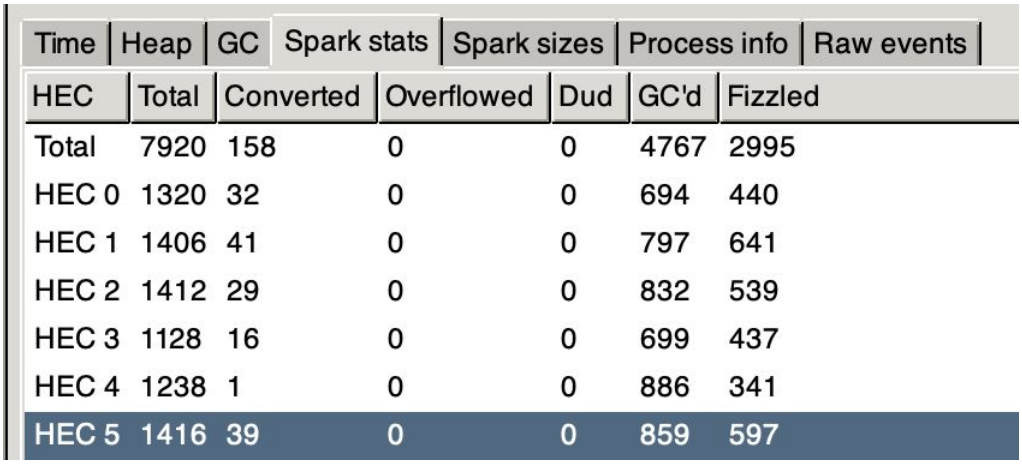
As seen in the data above, there is an improvement in performance as threads increase, however, not to the degree we were expecting. Looking at the 300 node graph sample set, we get a 1.5x best case performance increase, compared to 1.1x best case performance increase in the 1800 node graph case and a 1.4x best case performance increase in the 3000 node case.

The profiling below answered our question - the reason our performance wasn't increasing as expected was because of the "merge" function, where subgraphs are merged and colouring conflicts on adjacent nodes are resolved. As seen below, our intentional use of static partitioning was working as expected, threads are created as the graph is split into two subgraphs, however, our expectation was that our merge would occur on the parent thread that sparked the creation (i.e. as we work up the recursive tree). From the profiling below, it can be seen that merge only occurs on the initial execution thread.



There were many approaches we tried to solve this problem, however, none succeeded, leaving our performance increase at 1.3x (averaged over all graphs).. Our expectation is that by finding a way to merge on the parallel thread itself will bring us close to the 2x performance increase mark that we were expecting with the divide and conquer approach.

Another issue we noticed in our parallel solution were the number of sparks we were creating vs. converting. Only 2% of the sparks were being converted - our assumption is that this is also because of the lack of parallelism in the merge operation.



Time | Heap | GC | Spark stats | Spark sizes | Process info | Raw events

HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	7920	158	0	0	4767	2995
HEC 0	1320	32	0	0	694	440
HEC 1	1406	41	0	0	797	641
HEC 2	1412	29	0	0	832	539
HEC 3	1128	16	0	0	699	437
HEC 4	1238	1	0	0	886	341
HEC 5	1416	39	0	0	859	597

Appendix

I - References

[1] Gebremedhin, A. H. (1999). Parallel graph coloring. *UNIVERSITY M Thesis University of Bergen Norway Spring*.

[2] Fender, Walter. “The Graph Coloring Throwdown: Haskell vs. C++ vs. Java.” LinkedIn Engineering, Sept. 2011, engineering.linkedin.com/49/linkedin-coding-competitions-graph-coloring-haskell-c-and-java.

[3] Normann, Per. *Parallel Graph Coloring on Multi-Core CPUs* . June 2014, Paper available on: www.diva-portal.org/smash/get/diva2:730761/FULLTEXT01.pdf.

[4] Anderson, Loren. “ Edge Grundy Numbers of P3Pn and P3Cn.” *International Journal of Mathematics and Computer Science*, 8 Nov. 2015.

[5] Graph Coloring: Set 2 (Greedy Algorithm). (2018, May 1). Retrieved December 18, 2019, from https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/.

[6] m Coloring Problem: Backtracking-5. (2019, December 5). Retrieved December 18, 2019, from https://www.geeksforgeeks.org/m-coloring-problem-backtracking-5/.

[7] Induced subgraph. (2019, February 5). Retrieved December 18, 2019, from https://en.wikipedia.org/wiki/Induced_subgraph.

II - Graph Data Sets

Fender, Walter. “The Graph Coloring Throwdown: Haskell vs. C++ vs. Java.” LinkedIn Engineering, Sept. 2011, engineering.linkedin.com/49/linkedin-coding-competitions-graph-coloring-haskell-c-and-java.

Graphs available on: https://github.com/cheftako/LinkedInThreeColorability/tree/master/samples

III - Code

A repository of all our code, including test samples, can be found here:

install.sh - *This file installs all required dependencies*

```
stack install parallel

stack install monad-par

stack install random-shuffle
```

MakeFile - *This file compiles our project*

```
ifndef PNAME

override PNAME = graph_colouring.hs

endif


all:

            rm -rf output

            mkdir output

            stack ghc -- -O2 -threaded -rtsopts -eventlog $(PNAME)
```

Utils.hs - *This file contains utility functions shared across the project*

```

module Utils

( Node,

  Color,

  AdjList,

  Graph,

  response,

  wordsWhen,

  writeToFile,

  readGraphFile,

  isValidFile,

  getColor,

  getColors,
```

```
  getNeighbors,
  setColor,
  validColor,
  allVerticesColored,
  checkValidColored,
  checkValidColoredPar,
  colorAGraph,
  findClashingNodes,
  colorNode,
  setColors,
  getAllColors
) where

import qualified Data.Map as Map
import System.IO(Handle, hIsEOF, hGetLine, withFile,IOMode(ReadMode))
import Data.Maybe as Maybe
import Control.Parallel.Strategies (rpar, runEval)

-- Type to define a Graph as adjecncy list
type Node = String
type Color = Int
type AdjList = [Node]
type Graph = Map.Map(Node) (AdjList, Color)

colorAGraph :: FilePath -> (Graph -> Maybe Graph) -> String -> String -> IO String
colorAGraph graph_file algo outFolder inFolder = do
              let graph_file_name = last $ wordsWhen (=='/') graph_file
              let outFile = outFolder ++ "/" ++ graph_file_name ++ "_out"
              g <- readGraphFile $ inFolder ++ graph_file
              putStrLn ("coloring " ++ graph_file ++ " .. ")
              let output =  checkValidColored $ algo g
              let max_color = maximum $ getAllColors output
              response output graph_file
```

```
              writeToFile output max_color outFile
              return $ "done coloring " ++ graph_file ++ " with " ++ (show max_color)


-- check if <graph> from <fname> is {un}successfully colored and prints out a message
response :: Maybe Graph -> String -> IO ()
response graph fname = case graph of
                    Just _ -> putStrLn ("Successfully coloured graph " ++ fname)
                    Nothing ->  putStrLn ("Unable to colour graph "  ++ fname)


-- write <graph> to <fout>
writeToFile :: Maybe Graph -> Color -> String -> IO ()
writeToFile graph color fout = case graph of
                      Just a -> do writeFile fout ("true ncolors " ++ (show color) ++ "\n"++
printSolution a)
                      Nothing -> do writeFile fout "false\n"


readGraphFile :: String -> IO Graph
readGraphFile filename  = withFile filename ReadMode $ \handle -> loop handle readGraphLine
Map.empty


readGraphLine :: Handle -> Graph -> IO Graph
readGraphLine handle g = do args  <- (wordsWhen (==':')) <$> hGetLine handle
                            case args of
                              [node, adj] -> return $ Map.insert node (readAdjList adj, 0) g
                              _           -> return g


isValidFile :: FilePath -> Bool
isValidFile f =  "3color" /=  last ( wordsWhen (=='.') (show f) ) && f /= "." && f /= ".."


loop :: Handle -> (Handle -> Graph -> IO Graph) -> Graph -> IO Graph
loop h f g = do
              outgraph <- f h g
              eof <- hIsEOF h
```

```
              if eof then do return outgraph
              else do (loop h f outgraph) >>= (\y -> return y)


wordsWhen :: (Char -> Bool) -> String -> [String]
wordsWhen p s = case dropWhile p s of
                    "" -> []
                    s' -> w : wordsWhen p s''
                         where (w, s'') = break p s'
-- color of a vertex
-- e.g. color "A" g
getColor :: Node -> Graph -> Color
getColor n g = case Map.lookup n g of
                 Just v -> (snd v)
                 Nothing -> 0


-- given a list of nodes and a graph, retrieve all colour assignments to the node
getColors :: [Node] -> Graph -> [Color]
getColors [] _ = []
getColors (x:xs) g = getColor x g : getColors xs g


-- gets all neighbors for a node in a graph
getNeighbors :: Node -> Graph -> AdjList
getNeighbors n g = case Map.lookup n g of
                 Just v -> (fst v)
                 Nothing -> []


getAllColors :: Maybe Graph -> [Color]
getAllColors g = case g of
                  Just gr -> map (\(k, v) -> snd v) $Map.toList gr
                  Nothing -> []


-- assings a colour to a single node in the graph
setColor :: Graph -> Node -> Color -> Graph
```

```
setColor g n c = case Map.lookup n g of

                  Just v -> Map.insert n ((fst v), c) g

                  Nothing -> g




readAdjList :: String ->  AdjList

readAdjList x = wordsWhen (==',') x


-- checks if this color can be assigned to a vertex

-- e.g. validColor "A" 1 g

validColor :: Node -> Graph -> Color -> Bool

validColor n g c = c `notElem` getColors (getNeighbors n g) g


printSolution :: Graph -> String

printSolution g = unlines $ map (\n -> n ++ ':' : showColor n ) nodes

                  where nodes = Map.keys g

                        showColor n = show $ getColor n g


-- checks if all vertices have been coloured

-- e.g. allVerticesColored g

allVerticesColored :: Graph -> Bool

allVerticesColored g = 0 `notElem` getColors (Map.keys g) g


checkValidColoredPar :: Maybe Graph -> Maybe Graph

checkValidColoredPar g = case g of

                  Nothing -> Nothing

                  Just a -> checkValidColoredPar' (Map.keys a) a


checkValidColoredPar' :: [Node] -> Graph -> Maybe Graph

checkValidColoredPar' [] _ = Nothing

checkValidColoredPar' [n] g | getColor n g `notElem` getColors (getNeighbors n g) g = Just g

                      | otherwise = Nothing

checkValidColoredPar' nodes g
```

```
  | runEval $ do
      front <- rpar $ checkValidColoredPar' first g
      back <- rpar $ checkValidColoredPar' second g
      return (Maybe.isJust front && Maybe.isJust back) = Just g
  | otherwise = Nothing
  where (first, second) = splitAt (length nodes `div` 2) nodes


checkValidColored :: Maybe Graph -> Maybe Graph
checkValidColored g = case g of
                  Nothing -> Nothing
                  Just a -> checkValidColored' (Map.keys a) a


checkValidColored' :: [Node] -> Graph -> Maybe Graph
checkValidColored' [] g = Just g
checkValidColored' (n:ns) g
    | getColor n g `notElem` getColors (getNeighbors n g) g = checkValidColored' ns g
    | otherwise = Nothing



findClashingNodes :: Node -> Graph -> [Node]
findClashingNodes n g = [ x | x <- (getNeighbors n g), (getColor n g) == (getColor x g) ]



colorNode :: Node -> [Color] -> Graph -> Color
colorNode _ [] _ = 0
colorNode n (x:xs) g = if validColor n g x then do x
                            else do colorNode n xs g


setColors :: Graph -> [Node] -> Color -> Graph
setColors g [] _ = g
setColors g [n] c = setColor g n c
setColors g (n:ns) c = setColors (setColor g n c) ns c
```

GraphColoringAlgos.hs - *This file contains all the various graph colouring algorithms we came up with, including how we parallelized*

```
module GraphColoringAlgo
( backtracking,
  colorIndependent,
  divideConquerPar,
  greedy
) where


import Utils
import Data.List (sort)
import qualified Data.Map as Map
import Control.Parallel.Strategies (rpar, rseq, runEval, parListChunk, using, parMap,
parBuffer)


backtracking :: [Node] -> [Color] -> [Color] -> Graph -> Maybe Graph
backtracking _ [] _ g = Just g
backtracking [] _ _ g = Just g
backtracking _ _ [] _ = Nothing
backtracking nodes@(n:ns) colors (c:cs) g
      | validColor n g c = case (backtracking ns colors colors $ setColor g n c) of
                              Just gout -> Just gout
                              Nothing -> backtracking nodes colors cs g
      | otherwise = backtracking nodes colors cs g


greedy :: [Node] -> [Color] -> [Color] -> Graph -> Maybe Graph
greedy _ [] _ g = Just g
greedy [] _ _ g = Just g
greedy _ _ [] _ = Nothing
greedy nodes@(n:ns) colors (c:cs) g
      | validColor n g c = greedy ns colors colors $ setColor g n c
      | otherwise = greedy nodes colors cs g
```

```
inducedGraph :: Graph -> [Node] -> Graph
inducedGraph g nodes =  Map.fromList ( map (\x -> (x, (adj x, 0))) nodes `using` parListChunk
(length nodes `div` 2) rseq)
                          where adj = (\nx -> filter (\y -> y `elem` nodes) $ getNeighbors
nx g)



independentSet :: Graph -> Graph -> [Node] -> [Node] -> [Node]
independentSet _ _ [] i = i
independentSet g ig u@(x:_) i | length (Map.keys ig) == 0 = i
                              | otherwise = independentSet g ig_new u_new i_new
                                          where i_new = x : i
                                                u_new = filter (\y -> y `notElem` (x:
getNeighbors x g)) u
                                                ig_new = inducedGraph g  u_new


colorNodes :: Graph -> [Node] -> Color -> Graph
colorNodes g [] _ = g
colorNodes g nodes c = Map.union (fst pr) (Map.mapWithKey (\_ x -> (fst x, c)) (snd pr))
                       where pr = Map.partitionWithKey (\k _ -> k `notElem` nodes) g


colorIndependent :: Graph -> Graph -> [Node] -> [Color] -> Maybe Graph
colorIndependent g _ _ [] = Just g
colorIndependent g _ [] _ = Just g
colorIndependent g ig u (c:cs) | length (Map.keys ig) == 0 = Just g
                               | otherwise = runEval $ do
                                           i_new <- rseq $ independentSet ig ig u_nodes []
                                           u_new <- rpar $ filter (\y -> y `notElem` i_new)
u
                                           colored_g <- rpar $ colorNodes g i_new c
                                           ig_new <- rpar $ inducedGraph g u_new
```

```
                                          return $ colorIndependent colored_g ig_new u_new
cs
                                          where u_nodes = Map.keys ig


divideConquerPar :: [Node] -> Graph -> Maybe Graph
divideConquerPar n g = divideConquerPar' n [1..(length (Map.keys g))] g


divideConquerPar' :: [Node] -> [Color] -> Graph -> Maybe Graph
divideConquerPar' _ [] g = Just g
divideConquerPar' [] _ g = Just g
divideConquerPar' [n] colors g
  | allVerticesColored g = Just g
  | otherwise =
      if nodeColor > 0 then do
          Just $ setColor g n nodeColor
      else do
          Nothing
      where nodeColor = colorNode n colors g
divideConquerPar' nodes colors g
  | allVerticesColored g = Just g
  | otherwise = runEval $ do
    front <- rpar $ divideConquerPar' first colors $ subGraph first g Map.empty
    back <- rpar $ divideConquerPar' second colors $ subGraph second g Map.empty
    case (front, back) of
      (Just g, Nothing) -> return $ Just $ g
      (Nothing, Just g) -> return $ Just $ g
      (Just g, Just y)  -> return $ Just $ merge (Map.keys (Map.union g y)) colors $
Map.union g y
      _                 -> return $ Nothing
    where (first, second) = splitAt (length nodes `div` 2) nodes


merge :: [Node] -> [Color] -> Graph -> Graph
merge [] _ g = g
```

```
merge [x] colors g = setColors g (findClashingNodes x g) $ head updateColors
  where updateColors = filter (validColor x g) colors
merge (x:xs) colors g = merge xs updateColors $ setColors g (findClashingNodes x g) $ head
updateColors
  where updateColors = filter (validColor x g) colors


subGraph :: [Node] -> Graph -> Graph
subGraph [] _ x = x
subGraph [n] g x = Map.union x (Map.insert n ((getNeighbors n g), (getColor n g)) x)
subGraph (n:ns) g x = subGraph ns g (Map.union x (Map.insert n ((getNeighbors n g), (getColor
n g)) x))
```

graph_colouring.hs - *This is the main driver file for our solution*

```
-- stack ghc -- --make -Wall -O2 -threaded -rtsopts -eventlog graph_colouring.hs
import Utils
import GraphColoringAlgo


import System.Exit(die)
import System.Environment(getArgs, getProgName)


import qualified Data.Map as Map
import Control.Monad.IO.Class (liftIO)
import Control.Monad.Par.Combinator (parMapM)
import Control.Monad.Par.IO (runParIO)
import System.Directory
import System.Random.Shuffle


main :: IO ()
main = do
  args <- getArgs
  pn <- getProgName
```

```
  let errorMsg = "Usage: " ++ pn ++ " <input-{filename/foldername}> <number-of-colors> <algo:
{divide-conquer/backtracking/IndepSet/greedy}> <method: file/folder> <output-folder>"
  case args of
    [graph_file, number_colours, algo, method, outFolder] -> do
      func <- case algo of
        "backtracking" -> do
                let colours = read number_colours
                return (\g -> backtracking (Map.keys g) [1..colours] [1..colours] g)
        "indep-set" -> return (\g -> colorIndependent g g (Map.keys g) [1..])
        "greedy" -> return (\g -> backtracking (Map.keys g) [1..] [1..] g)
        "divide-conquer" -> do
          return (\g -> divideConquerPar (Map.keys g) g)
      case method of
        "file" -> do
          msg <- colorAGraph graph_file func outFolder ""
          putStrLn msg
        "folder" -> do
          let inFolder = graph_file
          filepaths <- filter isValidFile <$> getDirectoryContents inFolder
          filepathShuffled <- shuffleM filepaths
          putStrLn $ "coloring: \n" ++  (unlines filepathShuffled)
          responses <- runParIO $ parMapM (\f -> liftIO $ colorAGraph f func outFolder
inFolder) filepathShuffled
          mapM_ putStrLn responses
        _ -> do
          die errorMsg


    _ -> do
          die errorMsg
```