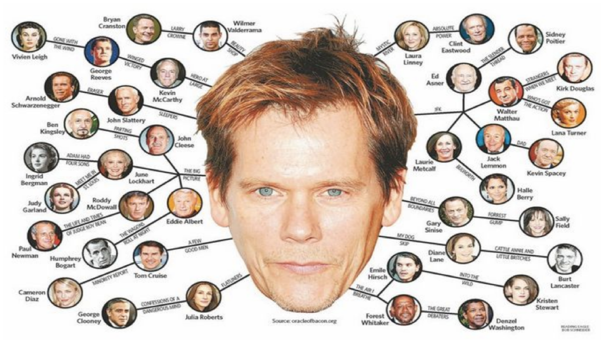
6 Degrees
Parallel Functional Programming Fall 2022

Jorge Raad - jar2356
Casey Olsen - ceo2132



1. Introduction

Inspired by the popular game, "Six Degrees of Kevin Bacon," our team implemented an algorithm that finds the minimum number of degrees (i.e. the shortest path) separating two actors, where actors who have appeared in the same movie are one degree apart from each other. The game claims that there is a maximum of six degrees of separation between any actor and Kevin Bacon, who has appeared in over 60 movies and co-starred with over 3,000 actors. We implemented an algorithm that finds the minimum degrees of separation between any two actors, which would be a generalization of this game.

We used a dataset from IMDb consisting of movies and actors that appear in them, but made our code general so that we can substitute the dataset in the case that we run into issues. In this case, we could substitute the individuals (in this case actors) and the groups to which they belong (in this case movies), or we could be even more general and have no distinction between these two entities (e.g. for movies vs. actors, you can imagine the sets of movies and actors as bipartite subgraphs of the same graph). This would allow us to model "degrees of separations" through other relationships like friendships.

2. Data Sets

2.1 IMDb data

We used free, public, IMDb datasets with over 12 million actors, movies and television shows. In particular we used the files:

Name.basics.tsv.gz - Associates actors, directors and cinematographers with the IDs of the titles they are most famous for

```
nconst primaryName    birthYear      deathYear      primaryProfession      knownForTitles
nm0000002,Lauren Bacall,1924,2014,actress,soundtrack,tt0071877,tt0037382,tt0038355,tt0117057
nm0000003,Brigitte Bardot,1934,\N,actress,soundtrack,tt0054452,tt0049189,tt0057345,tt0056404
nm0000004,John Belushi,1949,1982,actor,writer,tt0077975,tt0078723,tt0080455,tt0072562
nm0000005,Ingmar Bergman,1918,2007,writer,director,actor,tt0083922,tt0060827,tt0050986
```

Title.basics.tsv.gz - Associates IDs of titles with the actual name of the title

```
tconst,titleType,primaryTitleoriginalTitle,isAdult,startYear,endYear,runtimeMinutes,genres
tt0003362,movie,The Sea Wolf,The Sea Wolf,0,1913,\N,70,Drama
tt0003363,movie,The Seed of the Fathers,The Seed of the Fathers,0 1913,\N\N,Drama,War
tt0003364,short,The Shadow,The Shadow,0,1914,\N,\N,Drama,Short
tt0003365,movie,Shadows of the Moulin Rouge,Shadows of the Moulin Rouge,0,1913,\N,60,Drama
tt0003366,movie,The Shame of the Empire State,The Shame of the Empire State,0,1913,\N,Drama
```

2.2 Preprocessing

In order to make our program's job earlier, and based on the feedback from the teaching staff, we decided to preprocess our data into a more readable format. Because we only care about the actors and the titles they appear in, we filtered out the rest of the columns. Additionally, we translated the titles the actors starred in from IDs to names.

We wrote a python script that read the title data set and created an in-memory dictionary mapping title IDs to title names. We then processed the names file line-by-line dropping the unnecessary columns and translating the movie ids into movie titles and then writing the new output to a new file.

```python
# create an in memory title id to name mapping
title_dict = {}
title_file = open('title_subset_notv.csv', 'r')
title_lines = title_file.readlines()
for line in title_lines:
    line = line.rstrip()
    data = line.split('\t')
    title_dict[data[0]] = data[1]

new_file = open('movie_data_processed.csv', 'w')

# read the name file and translate the movie titles
with open('name.basics.tsv', 'r') as file:
    for line in file:
        data = line.rstrip().split('\t')
        new_titles = []
        for t in data[-1].split(','):
            if t in title_dict:
                new_titles.append(title_dict[t])
        if (len(new_titles) > 0):
            new_file.write(data[1] + '\t' + ','.join(new_titles) + '\n')
```

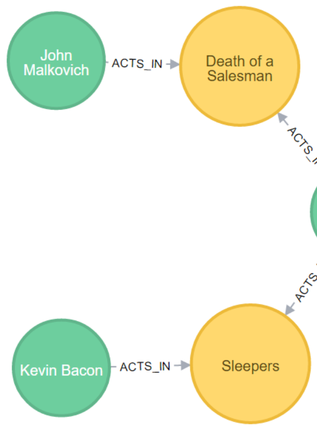
As a result, our movie data to feed into our program looks like the data snippet below. We used a dataset of 1 million actors, who each star in a number of movies. So when we combine the actors and titles, our adjacency list will have a few million entries.

```
Joan Fontaine  The Constant Nymph,Suspicion,Letter from an Unknown Woman,Rebecca
Clark Gable    Mutiny on the Bounty,Gone with the Wind,Red Dust,It Happened One Night
Judy Garland   Meet Me in St. Louis,The Wizard of Oz,Judgment at Nuremberg,A Star Is Born
John Gielgud   Murder on the Orient Express,Julius Caesar,Shine,Arthur
Jerry Goldsmith       Congo,The Boys from Brazil,L.A. Confidential,Star Trek: First Contact
Cary Grant     North by Northwest,Suspicion,Charade,Operation Petticoat
```

2.3 Generated Data

In order to explore the performance of our algorithm with various kinds of graphs, we created a Python script that randomly generates graphs with a given number of nodes and a given probability that an edge between two nodes exists. This allowed us to begin testing our BFS code on smaller datasets early on.

3. Graph Construction



3.1 Data Structure

We used an adjacency list to represent our graph. We used the Haskell Map class to store relationships between movies and actors. The key is a string type, which is either the name of an actor, or the name of a movie or tv series. The value is a set of strings. If the key is an actor, the value will be a set of movies, and likewise if the key is a movie, the value will be a set of actors. We use a set for fast lookup when we traverse the graph.

Haskell type of our graph data structure.

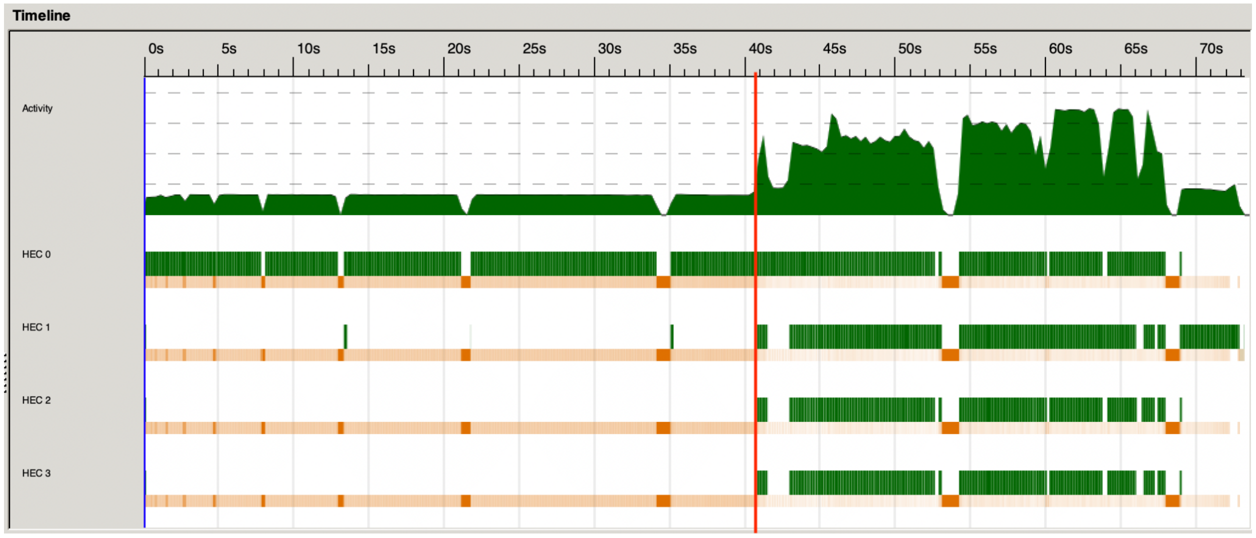
```
Map.Map String (Set String)
```

3.2 Sequential Construction

We first created our graph sequentially. We read in our movie data and recursively add entries to the map. For each row, we added a key for the actor and a set of all the movies they are associated with. Additionally, we added that actor to the set of the movie. This way we have association lists in both directions, and can do much faster lookups when we traverse the graph.

```
parseData :: String -> IO (Map.Map String (Set String))
parseData filename = do
    contents <- readFile filename
    return (buildGraph (lines contents) Map.empty)

buildGraph :: [String] -> Map.Map String (Set String) -> Map.Map String (Set String)
buildGraph [] graph = graph
buildGraph (x:xs) graph = buildGraph xs graphWMovies
    where graphWMovies = foldr (\movie g -> Map.insertWith (Set.union) movie (Set.singleton
name) g) graphWActor movies
          graphWActor = Map.insert name (Set.fromList movies) graph -- insert actor keys
          movies = splitOn "," movieStr
          [name, movieStr] = splitOn "\t" x
```

When we ran threadscope to analyze our program with this graph construction, we can clearly see only one thread is active in the construction process. The whole process took about 40 seconds as we can see by the red line marking the transition from graph construction to graph traversal.



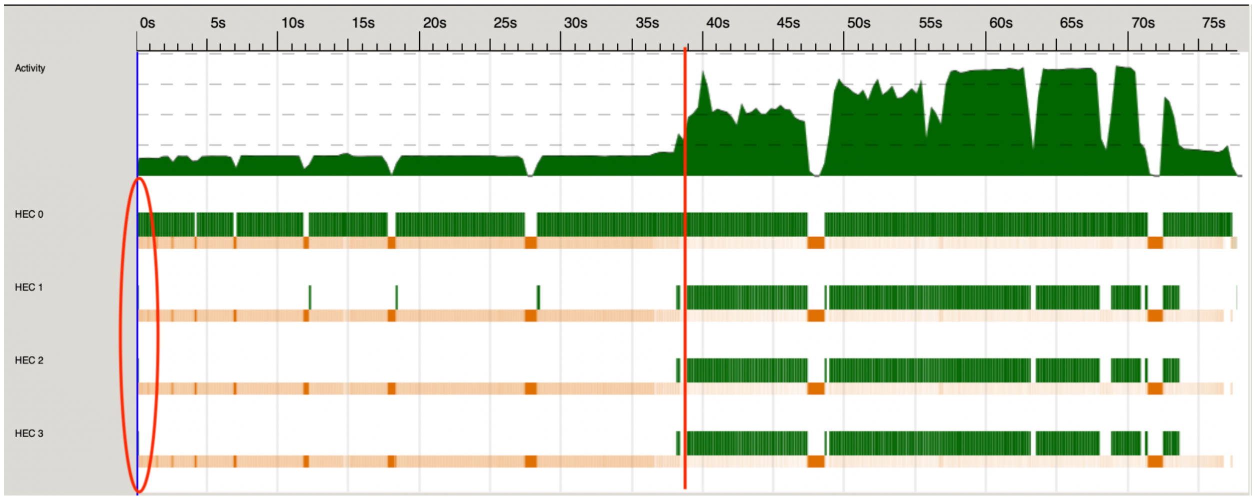
3.3 Parallel Construction

3.3.1 Control.Parallel.Strategies Library

To speed up our graph construction, we tried to leverage the Control.Parallel.Strategies library to parallelize the I/O. We split our dataset into four files of approximately equal length and did our graph construction algorithm on each of them in parallel. We then merged pairs of resulting maps together in parallel, and finally returned one map that is the union of each of the individual files.

```
import Control.Parallel.Strategies( runEval, rpar, rseq )

readDataParallel :: String -> IO (Map.Map String (Set String))
readDataParallel filenamePrefix = runEval $ do
    d1 <- rpar (parseData $ filenamePrefix ++ "1.csv")
    d2 <- rpar (parseData $ filenamePrefix ++ "2.csv")
    d3 <- rpar (parseData $ filenamePrefix ++ "3.csv")
    d4 <- rpar (parseData $ filenamePrefix ++ "4.csv")
    _ <- rseq d1
    _ <- rseq d2
    _ <- rseq d3
    _ <- rseq d4
    m1 <- rpar (Map.unionWith Set.union <$>  d1  <*>  d2)
    m2 <- rpar (Map.unionWith Set.union <$>  d3  <*>  d4)
    _ <- rseq m1
    _ <- rseq m2
    return (Map.unionWith Set.union <$>  m1  <*>  m2)
```

Our threadscope graph showed that our parallelization attempts did not have an effect on the runtime of our program. We can see all of the I/O was done by the first thread and took approximately the same amount of time as the sequential read of a single file. If we zoom in at the very start, we can see each thread was initialized, but the work was not shared. We see three small segments of work on the second thread, which logically, must be where we switched between each of the four files.



3.3.2 Control.Concurrent.Async Library

We made an additional attempt at parallel I/O using the Control.Concurrent.Async library in hopes it could actually split our file reading across threads.

```
import Control.Concurrent.Async( concurrently )

readDataParallel filenamePrefix = do
    (d1, d2) <- concurrently (parseData $ filenamePrefix ++ "1.csv") (parseData $
filenamePrefix ++ "2.csv")
    (Map.unionWith Set.union d1 d2)
```

However, we found the same results as the strategies library and this did not help the execution time of our program. We concluded that I/O is hard to perform in parallel and decided to focus on the parallelization of our BFS algorithm.

4. BFS Traversal

4.1 Sequential

Because our project is concerned with finding the number of actors between two specified actors, our data can be represented as a graph of edges without weights. Since the edges of our graph are unweighted, the fastest algorithm to find the shortest path between two actors would be Breadth First Search (BFS). However, a standard BFS implementation leaves us with a problem: it entails repeatedly removing the next vertex from a FIFO queue, checking its neighbors, and then adding its neighbors that have yet to be visited to the queue. This presents us with a problem as adding and popping vertices to and from the queue in parallel would result in a traversal through the nodes of the graph in an order that may not be in order of increasing distance. This is a problem as the BFS implementation relies on the fact that when a vertex has been removed from the queue, there are no previous vertices that could be farther away from the source and there are no unexplored vertices that can be closer than it to the source. However, if vertices that are at different distances from the source were to be removed and processed simultaneously by multiple threads, one can imagine a situation where, a common neighbor node *A* is first added to the shared queue by the farther node, resulting in the distance of vertex *A* (and any subsequent neighbors of *A*) being set to an incorrect value.

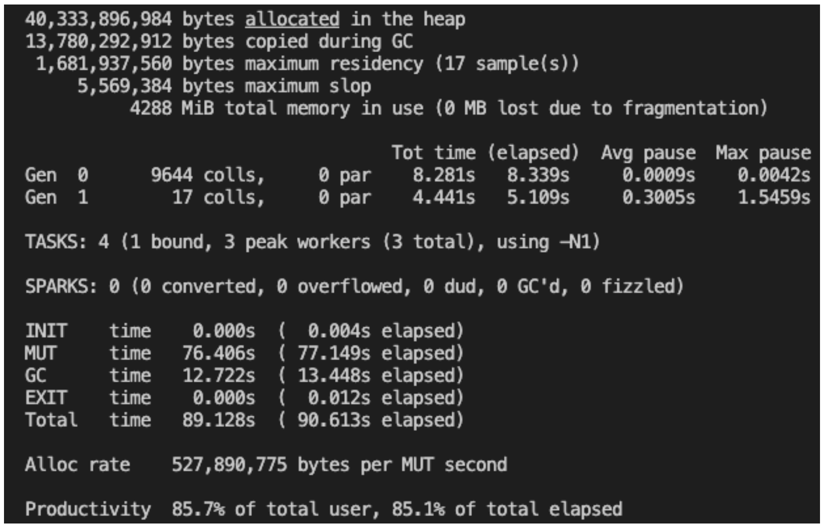
Because of this, we started off by implementing our sequential algorithm for BFS by having our algorithm explore nodes level by level.

Pseudo Code:

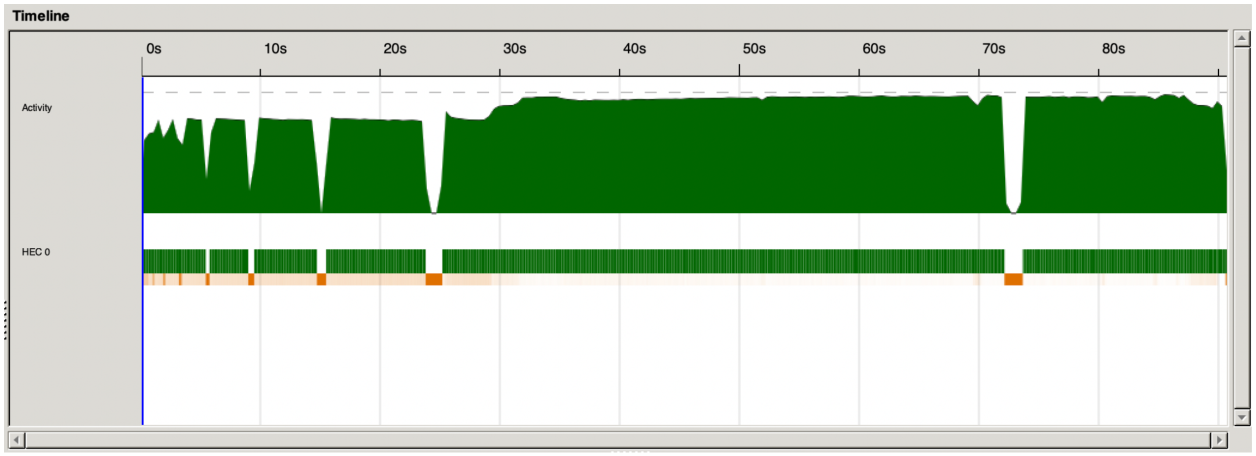
```
runBFS(V, E, s):
    distances = emptyArray()
    distance[s] = 0
    currentLevel = [s]
    levelNum = 0
    while currentLevel is not empty:
        nextLevel = []
        for u in nextLevel:
            for (u, v) in E:
                if v not in visited:
                    dist[v] = levelNum + 1
                    nextLevel.insert(v)
        levelNum++
        currentLevel = nextLevel
```

Where this level-by-level approach differs from the standard BFS implementation is that, instead of picking the next vertex to process by always popping it from the same FIFO queue, which can result in the concurrency issue discussed above if we attempt to parallelize it, the algorithm only deals with vertices at the same level (i.e., the same distance away from the source). When this code is parallelized, the previously mentioned problem is no longer an issue as, even if there is a common neighbor between two nodes being processed simultaneously, it does not matter which one sets the distance of the common node *A*; the distance would be the same in either case since the two vertices are the same distance *d* away from the source and the common node is just one edge away from both, so regardless of which vertex the path to *A* goes through, its distance would still be $d + 1$.

These are the results from running this algorithm sequentially on the data:

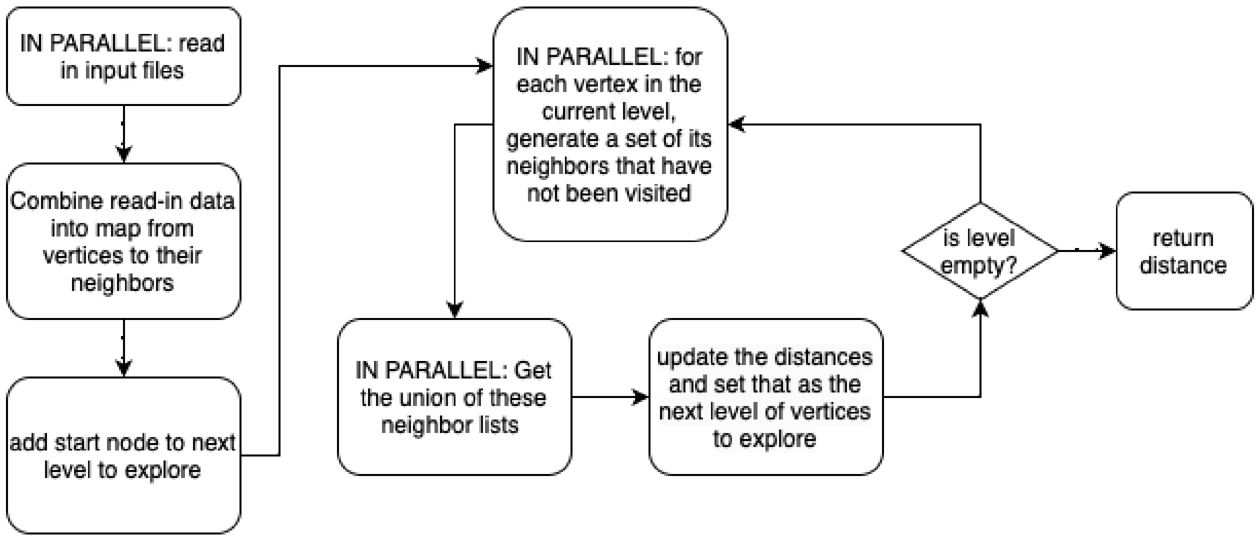


```
  40,333,896,984 bytes allocated in the heap
  13,780,292,912 bytes copied during GC
   1,681,937,560 bytes maximum residency (17 sample(s))
       5,569,384 bytes maximum slop
            4288 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      9644 colls,     0 par    8.281s   8.339s     0.0009s    0.0042s
  Gen  1        17 colls,     0 par    4.441s   5.109s     0.3005s    1.5459s

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.000s  (  0.004s elapsed)
  MUT     time   76.406s  ( 77.149s elapsed)
  GC      time   12.722s  ( 13.448s elapsed)
  EXIT    time    0.000s  (  0.012s elapsed)
  Total   time   89.128s  ( 90.613s elapsed)

  Alloc rate    527,890,775 bytes per MUT second

  Productivity  85.7% of total user, 85.1% of total elapsed
```



4.2 Parallelization Approach

As discussed in the previous section, implementing our BFS algorithm so that it processes vertices level by level allows us to more easily parallelize the exploration of our graph. Our first goal was to parallelize the retrieval of the neighbors for each of the vertices in the current level. This involved taking the set of the vertices to be explored in the current level, and then generating a set of each vertex's unvisited neighbors in parallel using different strategies. After this is completed for all of the vertices in the current level, we are left with a list of sets of neighbors for each of the vertices. Following this, we need to combine these results from the previous part in order to create the next set of vertices to be visited in the next level. We aimed to parallelize this combination by breaking it up. We split up the resulting list of sets of neighbors into smaller chunks, finding the union of these smaller chunks and then finding the union of the resulting unions. After this, the distances for each of the processed nodes are updated sequentially, and we move onto exploring the next level.

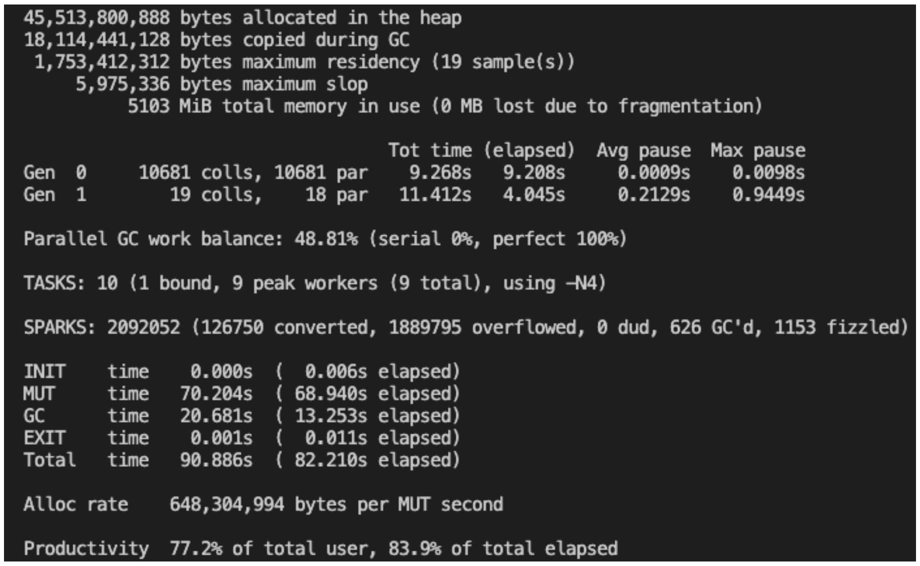

IN PARALLEL: read in input files
Combine read-in data into map from vertices to their neighbors
add start node to next level to explore
IN PARALLEL: for each vertex in the current level, generate a set of its neighbors that have not been visited
IN PARALLEL: Get the union of these neighbor lists
update the distances and set that as the next level of vertices to explore
is level empty?
return distance

4.3 Using `parList`

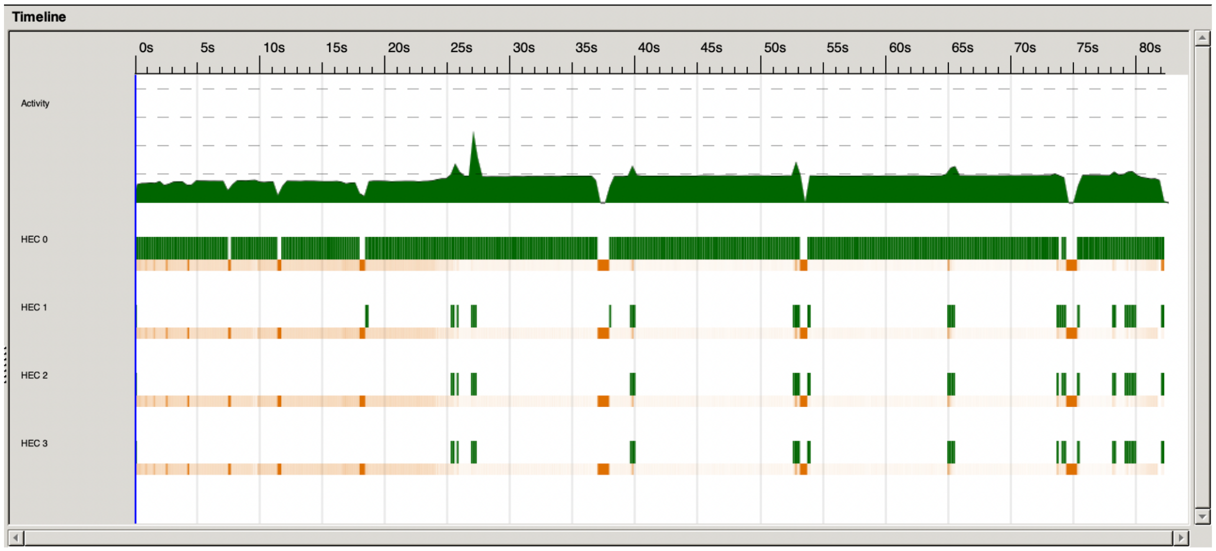
The first strategy we tried using when generating the sets of each vertex's unvisited neighbors in parallel was converting the `Set` of vertices to process into a `List` and then using `parList`, passing in `rdeepseq` as the `Strategy`.

While this did somewhat speed up the program, it unsurprisingly resulted in the majority of sparks going unconverted because of overflow as a result of creating two many sparks at once.

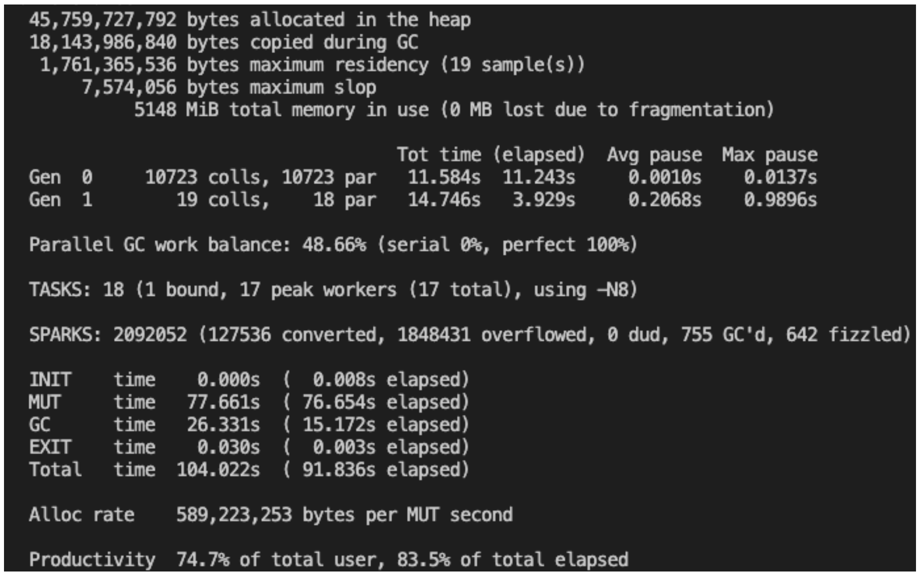
These are the results from running the algorithm on 4 cores:



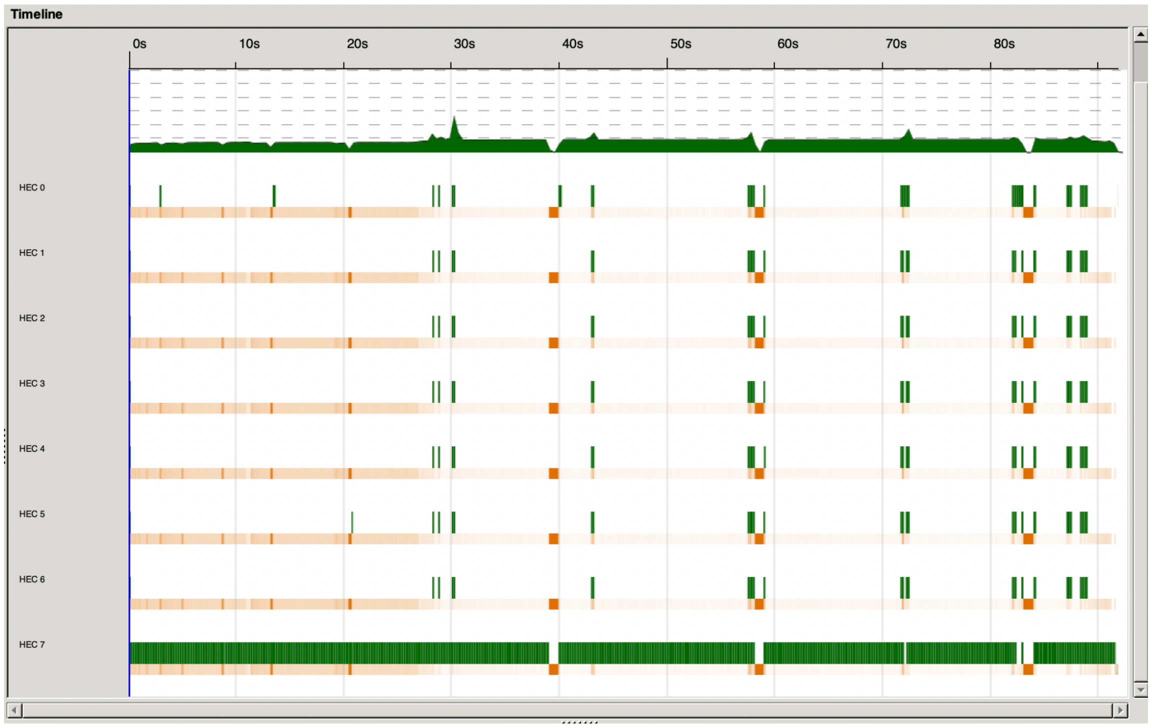
```
45,513,800,888 bytes allocated in the heap
18,114,441,128 bytes copied during GC
 1,753,412,312 bytes maximum residency (19 sample(s))
     5,975,336 bytes maximum slop
          5103 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0     10681 colls, 10681 par    9.268s    9.208s     0.0009s    0.0098s
Gen  1        19 colls,    18 par   11.412s    4.045s     0.2129s    0.9449s

Parallel GC work balance: 48.81% (serial 0%, perfect 100%)

TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

SPARKS: 2092052 (126750 converted, 1889795 overflowed, 0 dud, 626 GC'd, 1153 fizzled)

INIT    time    0.000s  (  0.006s elapsed)
MUT     time   70.204s  ( 68.940s elapsed)
GC      time   20.681s  ( 13.253s elapsed)
EXIT    time    0.001s  (  0.011s elapsed)
Total   time   90.886s  ( 82.210s elapsed)

Alloc rate    648,304,994 bytes per MUT second

Productivity  77.2% of total user, 83.9% of total elapsed
```



These are the results from running the algorithm on 8 cores:



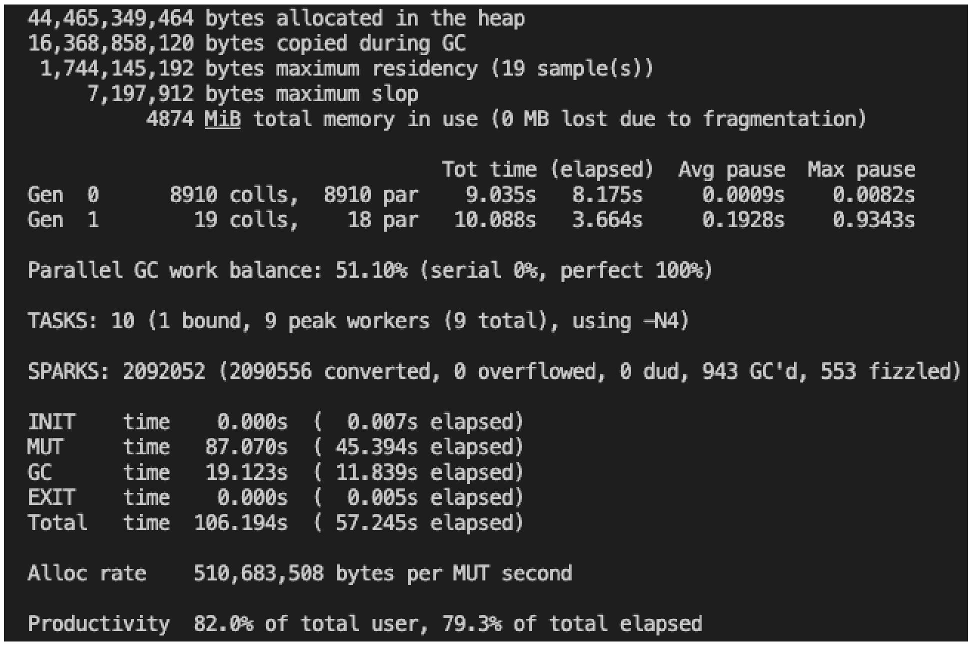
```
  45,759,727,792 bytes allocated in the heap
  18,143,986,840 bytes copied during GC
   1,761,365,536 bytes maximum residency (19 sample(s))
       7,574,056 bytes maximum slop
            5148 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     10723 colls, 10723 par   11.584s  11.243s     0.0010s    0.0137s
  Gen  1        19 colls,    18 par   14.746s   3.929s     0.2068s    0.9896s

  Parallel GC work balance: 48.66% (serial 0%, perfect 100%)

  TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

  SPARKS: 2092052 (127536 converted, 1848431 overflowed, 0 dud, 755 GC'd, 642 fizzled)

  INIT    time    0.000s  (  0.008s elapsed)
  MUT     time   77.661s  ( 76.654s elapsed)
  GC      time   26.331s  ( 15.172s elapsed)
  EXIT    time    0.030s  (  0.003s elapsed)
  Total   time  104.022s  ( 91.836s elapsed)

  Alloc rate    589,223,253 bytes per MUT second

  Productivity  74.7% of total user, 83.5% of total elapsed
```



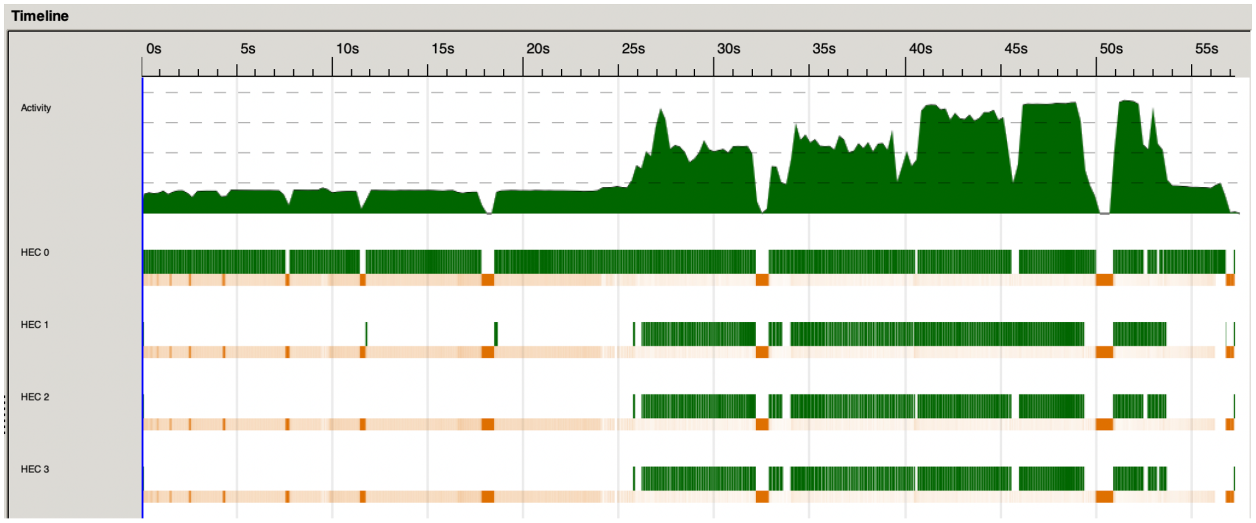
4.4 Using `parBuffer`

We tried to address this overflow issue by using parBuffer instead of parList. After experimenting with the size of the buffer, we arrived at 7500, which resulted in the creation of a large number of sparks to do parallel work without creating so many sparks simultaneously that they overflow. We are able to set such a large number of sparks because the amount of work

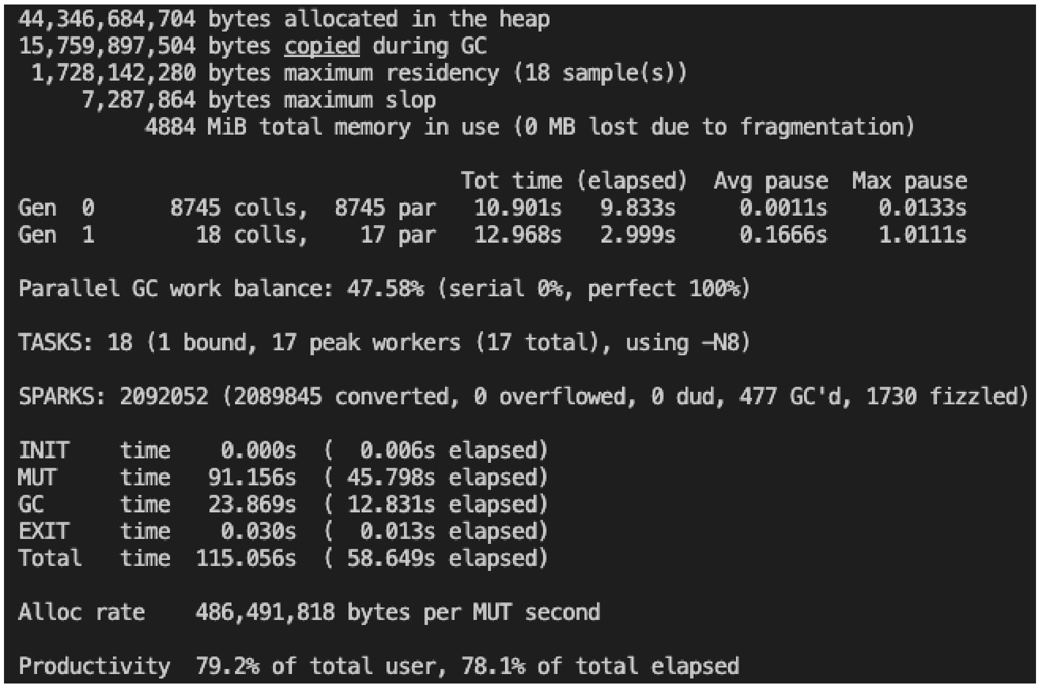
done by each (i.e., looking up the neighbors of a vertex in a map and removing the previously visited nodes from it) is very small.
This is the performance on 4 cores:



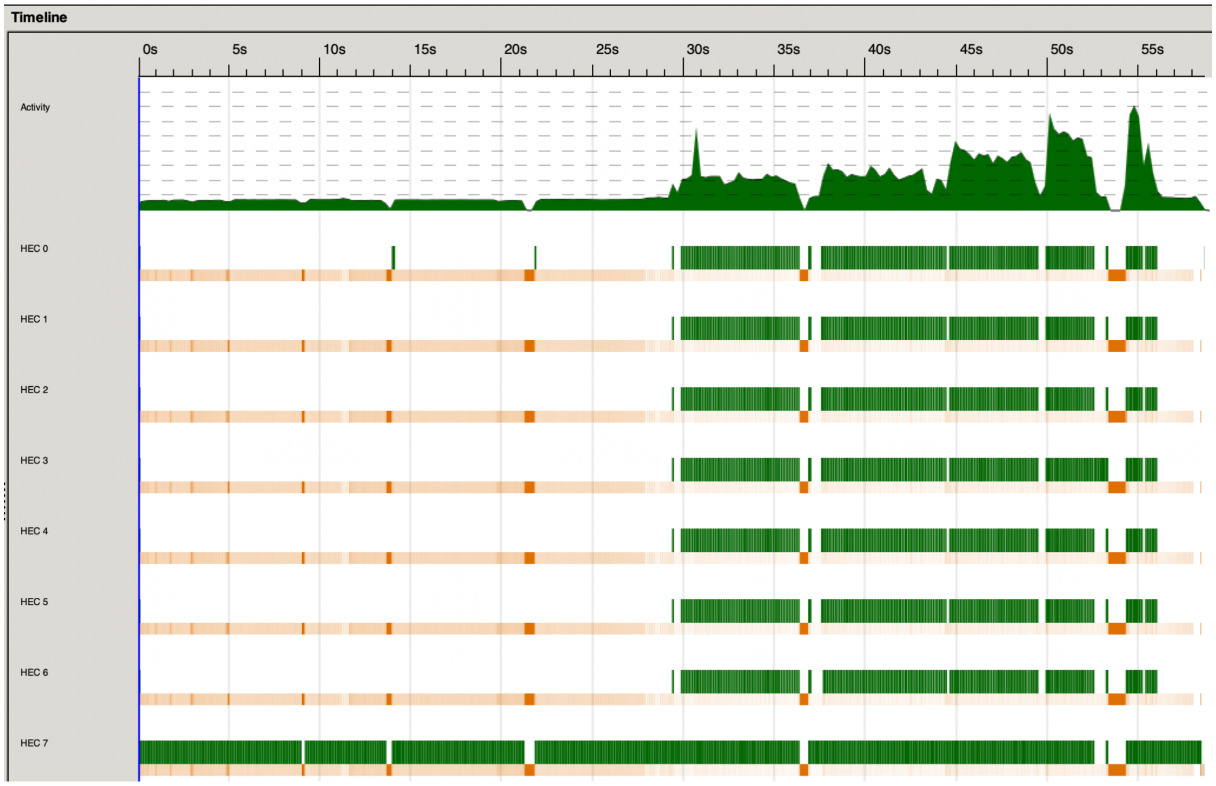
```
  44,465,349,464 bytes allocated in the heap
  16,368,858,120 bytes copied during GC
   1,744,145,192 bytes maximum residency (19 sample(s))
       7,197,912 bytes maximum slop
            4874 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      8910 colls,  8910 par    9.035s   8.175s     0.0009s    0.0082s
  Gen  1        19 colls,    18 par   10.088s   3.664s     0.1928s    0.9343s

  Parallel GC work balance: 51.10% (serial 0%, perfect 100%)

  TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

  SPARKS: 2092052 (2090556 converted, 0 overflowed, 0 dud, 943 GC'd, 553 fizzled)

  INIT    time    0.000s  (  0.007s elapsed)
  MUT     time   87.070s  ( 45.394s elapsed)
  GC      time   19.123s  ( 11.839s elapsed)
  EXIT    time    0.000s  (  0.005s elapsed)
  Total   time  106.194s  ( 57.245s elapsed)

  Alloc rate    510,683,508 bytes per MUT second

  Productivity  82.0% of total user, 79.3% of total elapsed
```



This is the performance on 8 cores:



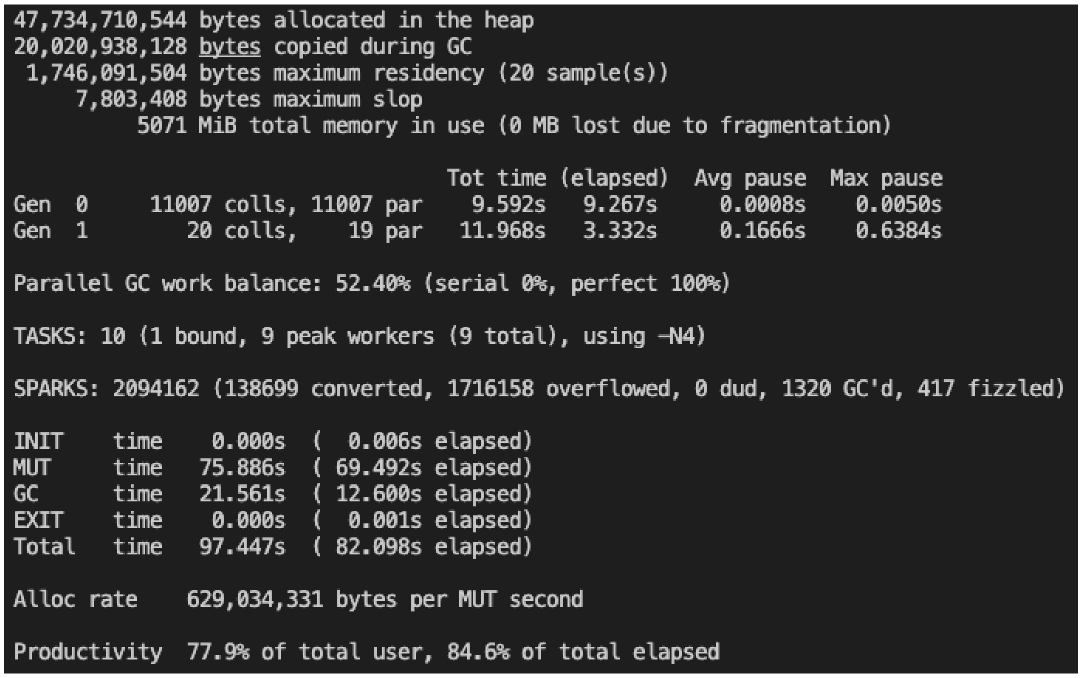
```
44,346,684,704 bytes allocated in the heap
15,759,897,504 bytes copied during GC
 1,728,142,280 bytes maximum residency (18 sample(s))
     7,287,864 bytes maximum slop
          4884 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0      8745 colls,  8745 par   10.901s    9.833s     0.0011s    0.0133s
Gen  1        18 colls,    17 par   12.968s    2.999s     0.1666s    1.0111s

Parallel GC work balance: 47.58% (serial 0%, perfect 100%)

TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

SPARKS: 2092052 (2089845 converted, 0 overflowed, 0 dud, 477 GC'd, 1730 fizzled)

INIT    time    0.000s  (  0.006s elapsed)
MUT     time   91.156s  ( 45.798s elapsed)
GC      time   23.869s  ( 12.831s elapsed)
EXIT    time    0.030s  (  0.013s elapsed)
Total   time  115.056s  ( 58.649s elapsed)

Alloc rate    486,491,818 bytes per MUT second

Productivity  79.2% of total user, 78.1% of total elapsed
```



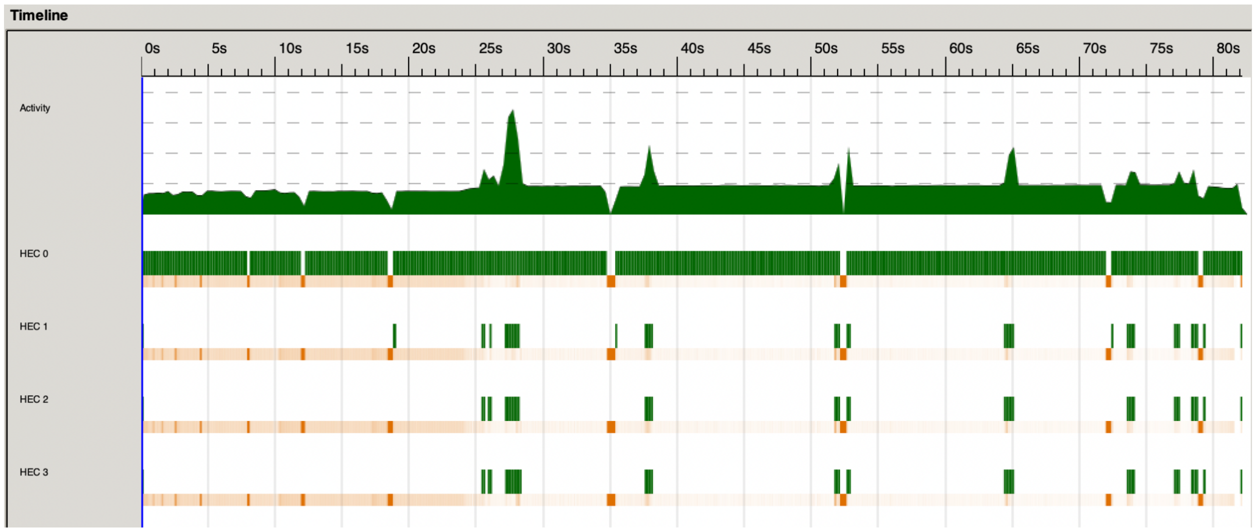
4.5 Parallelizing Union of Neighbor Lists

After the previous step of finding the sets of neighbors of each vertex at the current level, we have to find the union of these vertices. Our next goal was to break this up into parts that can be done in parallel. We did this by using List.chunksOf to split up the list of sets into smaller lists of sets. We then mapped the Set.unions function onto each of the lists of sets in parallel using parList. Finally, we take the union of these resulting sets sequentially and return the final set of vertices. This resulted in a lot of sparks overflowing.

Here is the performance on 4 cores:

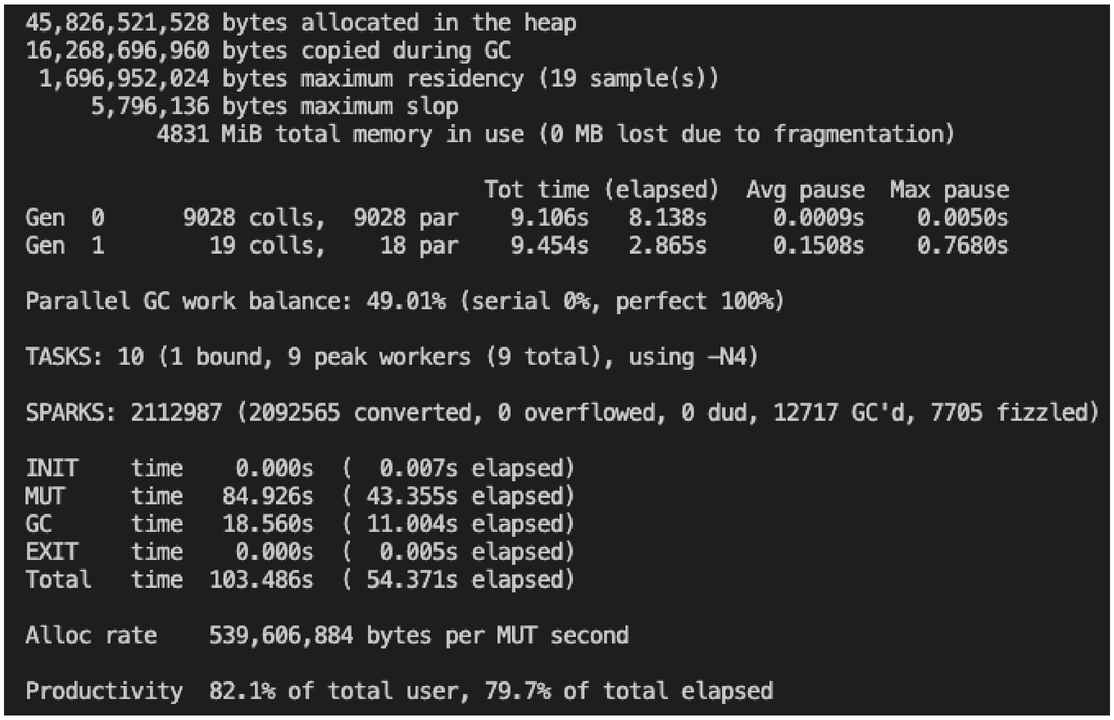


```
47,734,710,544 bytes allocated in the heap
20,020,938,128 bytes copied during GC
 1,746,091,504 bytes maximum residency (20 sample(s))
     7,803,408 bytes maximum slop
          5071 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0     11007 colls, 11007 par    9.592s   9.267s     0.0008s    0.0050s
Gen  1        20 colls,    19 par   11.968s   3.332s     0.1666s    0.6384s

Parallel GC work balance: 52.40% (serial 0%, perfect 100%)

TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

SPARKS: 2094162 (138699 converted, 1716158 overflowed, 0 dud, 1320 GC'd, 417 fizzled)

INIT    time    0.000s  (  0.006s elapsed)
MUT     time   75.886s  ( 69.492s elapsed)
GC      time   21.561s  ( 12.600s elapsed)
EXIT    time    0.000s  (  0.001s elapsed)
Total   time   97.447s  ( 82.098s elapsed)

Alloc rate    629,034,331 bytes per MUT second

Productivity  77.9% of total user, 84.6% of total elapsed
```

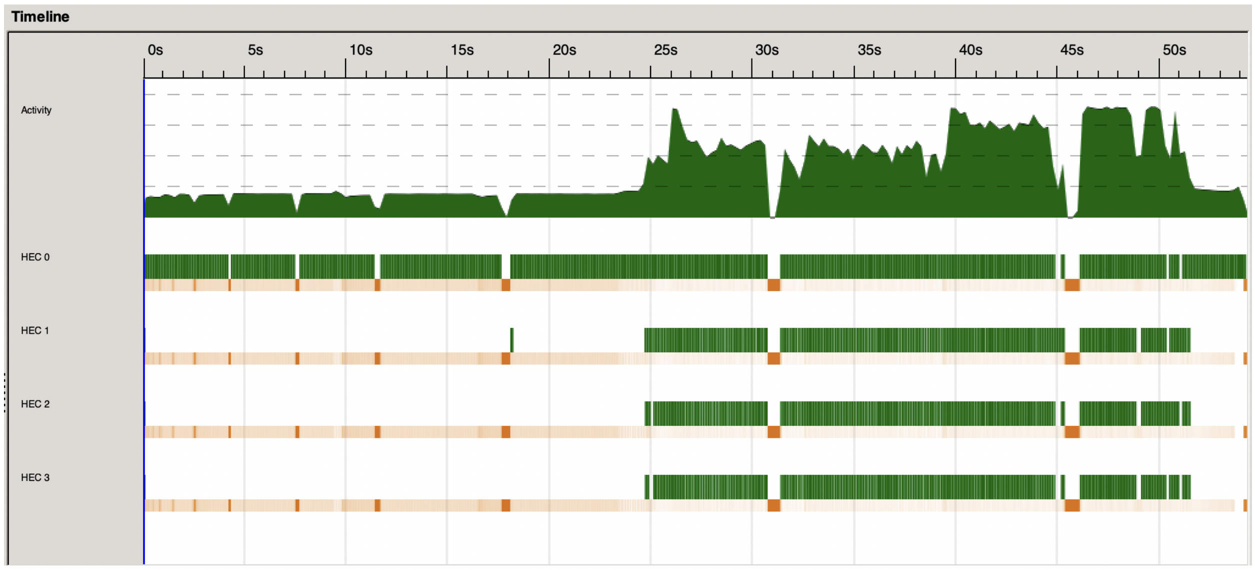


So, as we did in the previous part, we tried using parBuffer. We experimented with the size of the chunks and the parBuffer size, ultimately arriving at 64 and 8, respectively, which maximized parallelization while eliminating overflow. However, this did not have any noticeable speedup over the version with the fully sequential combination of the sets of neighbors.

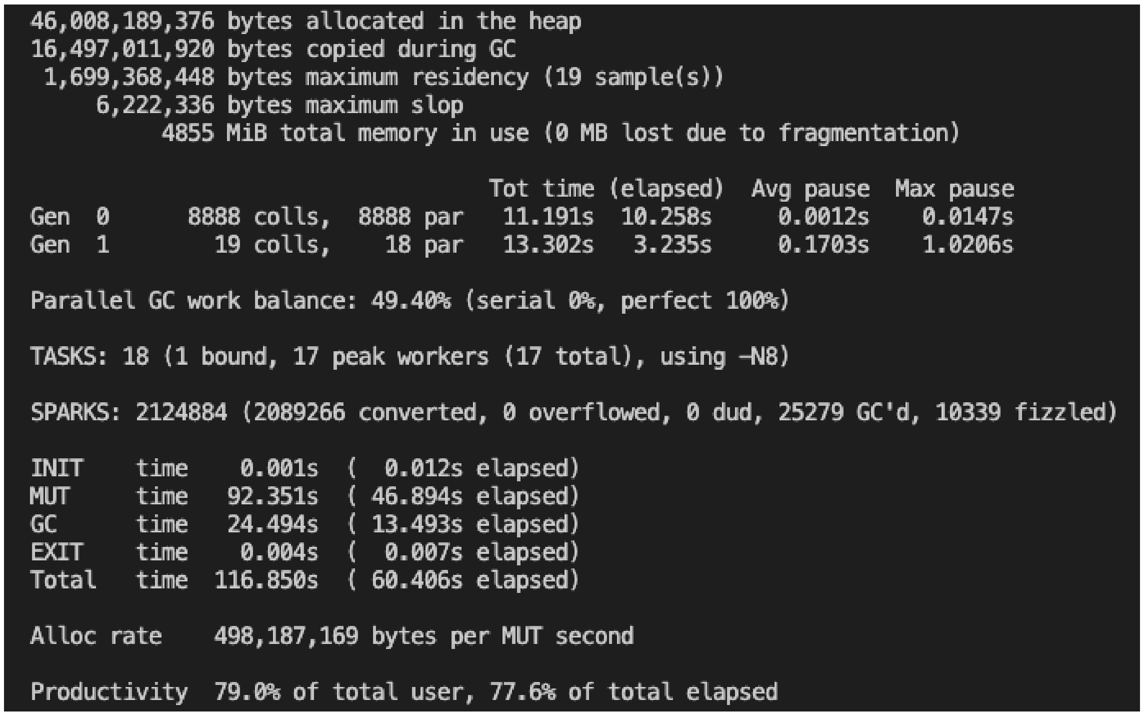
Here is the performance on 4 cores:



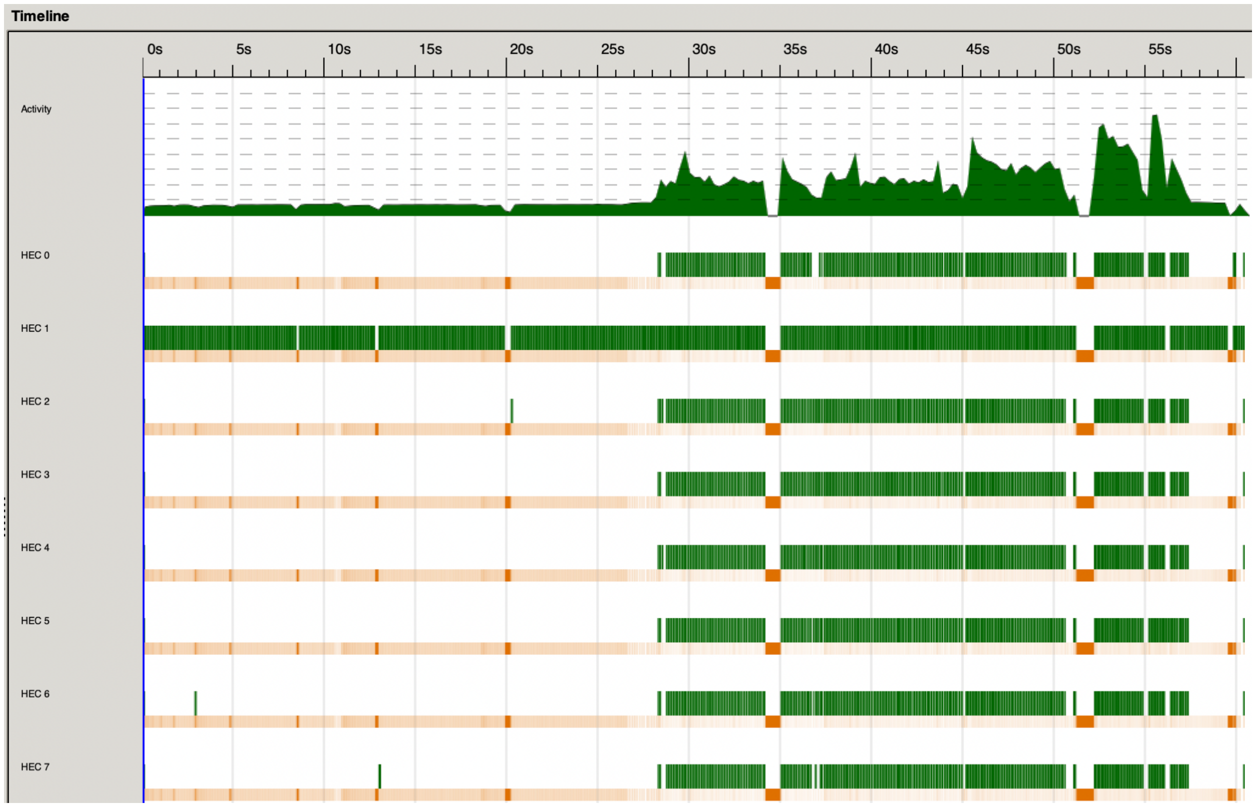
```
45,826,521,528 bytes allocated in the heap
16,268,696,960 bytes copied during GC
 1,696,952,024 bytes maximum residency (19 sample(s))
     5,796,136 bytes maximum slop
          4831 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0      9028 colls,  9028 par     9.106s   8.138s     0.0009s    0.0050s
Gen  1        19 colls,    18 par     9.454s   2.865s     0.1508s    0.7680s

Parallel GC work balance: 49.01% (serial 0%, perfect 100%)

TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

SPARKS: 2112987 (2092565 converted, 0 overflowed, 0 dud, 12717 GC'd, 7705 fizzled)

INIT    time    0.000s  (  0.007s elapsed)
MUT     time   84.926s  ( 43.355s elapsed)
GC      time   18.560s  ( 11.004s elapsed)
EXIT    time    0.000s  (  0.005s elapsed)
Total   time  103.486s  ( 54.371s elapsed)

Alloc rate    539,606,884 bytes per MUT second

Productivity  82.1% of total user, 79.7% of total elapsed
```



Here is the performance on 8 cores:



```
46,008,189,376 bytes allocated in the heap
16,497,011,920 bytes copied during GC
 1,699,368,448 bytes maximum residency (19 sample(s))
     6,222,336 bytes maximum slop
          4855 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0      8888 colls,  8888 par   11.191s  10.258s     0.0012s    0.0147s
Gen  1        19 colls,    18 par   13.302s   3.235s     0.1703s    1.0206s

Parallel GC work balance: 49.40% (serial 0%, perfect 100%)

TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

SPARKS: 2124884 (2089266 converted, 0 overflowed, 0 dud, 25279 GC'd, 10339 fizzled)

INIT    time    0.001s  (  0.012s elapsed)
MUT     time   92.351s  ( 46.894s elapsed)
GC      time   24.494s  ( 13.493s elapsed)
EXIT    time    0.004s  (  0.007s elapsed)
Total   time  116.850s  ( 60.406s elapsed)

Alloc rate    498,187,169 bytes per MUT second

Productivity  79.0% of total user, 77.6% of total elapsed
```



4.6 Reducing Sequential Work

At first we were not satisfied with the amount of work that has to be done sequentially between the parallel parts. When testing this algorithm on large datasets using multiple cores, we saw a distinct pattern in the parallelization of our program. The parallel parts of the code saw great utilization of the multiple cores and thus a speedup. However, there is some sequential code that must run between the parallel parts in order to combine the results and determine the vertices to be visited in the next level. As we studied the parallelization and realized that the sequential work that happens between each parallel part is actually relatively large. This was a concern because, as we know from Amdahl's Law, the more sequential code there is, the less we can speed up the program using parallelization.

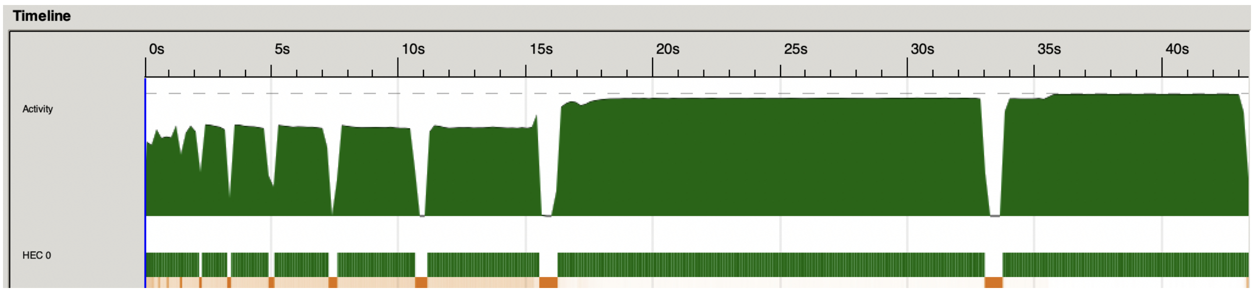
We explored further reducing the amount of time the sequential part takes but realized we could not improve this much. For example, we currently use sets when keeping track of groups of vertices (e.g., to store visited nodes, each vertex's neighbors, and the next level of vertices to visit). This allows us to efficiently take unions and differences between sets, which is necessary when determining which vertices to explore next and removing vertices that have already been visited. However, our use of sets means that we currently have to transform our set of vertices at the current level into a list before using parBuffer on it, which is O(n). This is because functions like parList, parBuffer, and parTraversable only function on Traversables, which Lists are, but Sets are not. We also could not implement a similar "parSet" function since there is no way to pop random elements from a set. Additionally, the part in the rest of the code used to process the current level is at least O(n) anyways as we must sequentially update the distances of all visited nodes in each iteration. Additionally, replacing our implementation using sets with a list-based implementation dramatically increases the run time (we had to terminate the process) due to much less inefficient runtime for finding set unions and differences.

4.7 Sparse vs. Dense

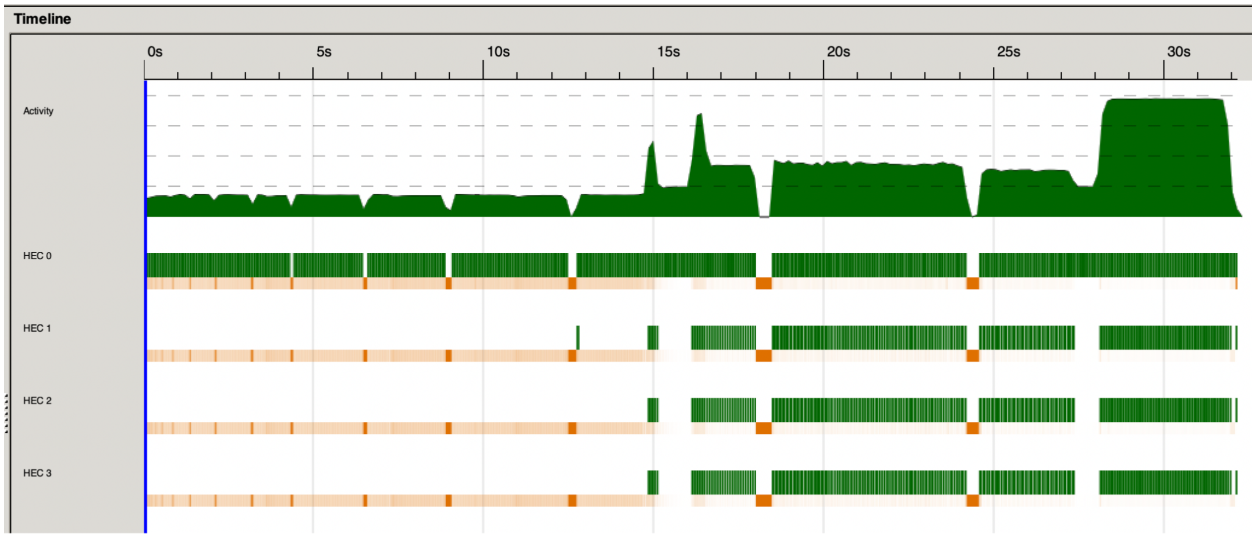
We also explored the performance of our code on different kinds of graphs. For example, here are the results for a randomly generated graph with 40,000 nodes and a 0.05% chance that an edge between any two vertices exists.

100,000 vertices, 0.05% any two edges are connected

1 core: 43.382s elapsed

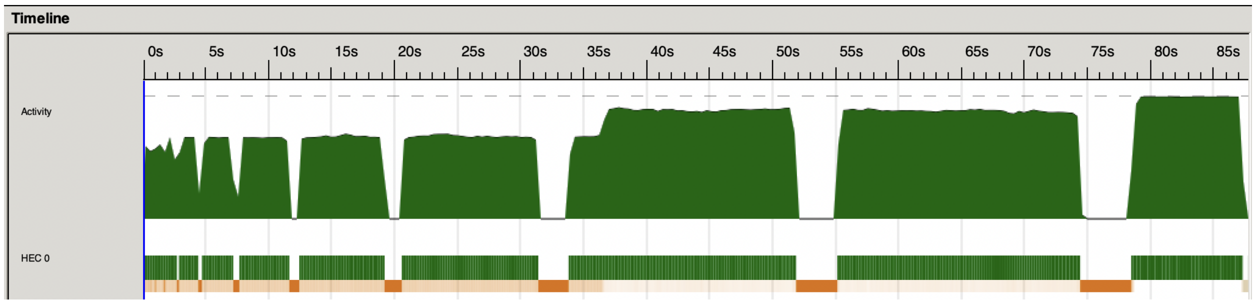


4 cores: 32.161s elapsed (1.35x speedup)

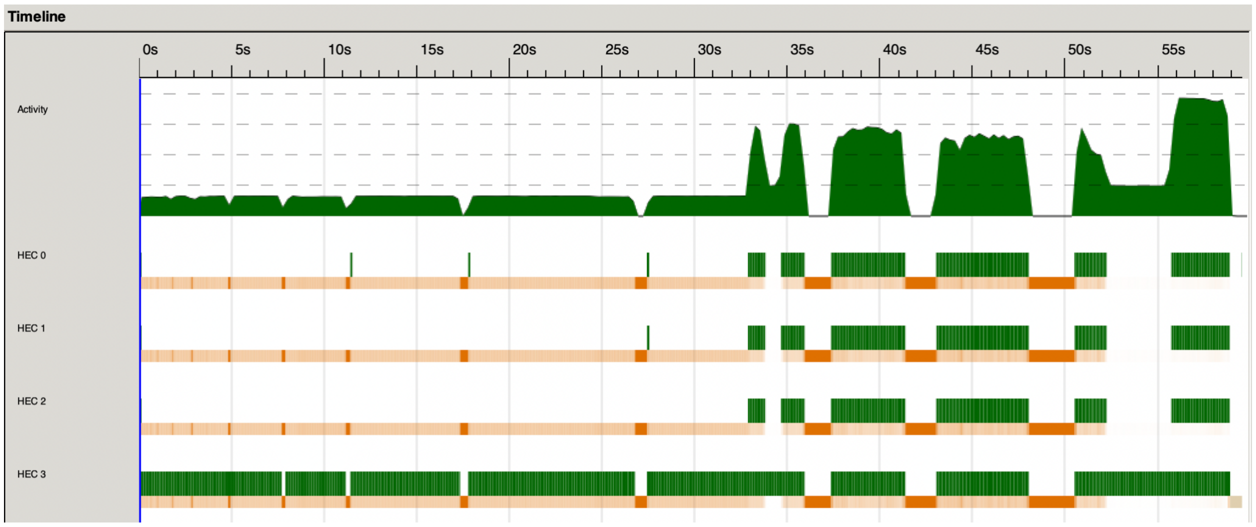


20,000 vertices, 5% any two edges are connected

1 core: 87.722s



4 cores: 59.525s elapsed (1.47x speedup)



4.8 Number of Cores

We ran the final product on the actor data using different numbers of cores and we arrived at the following numbers. Because the time it takes to read in the file does not vary and takes about 20 seconds each run, we have introduced two columns to attempt to better visualize the speedup achieved for just the BFS part of the code. The "approximate BFS time" was taken by subtracting 20 seconds from the total time.

As expected, adding additional cores did reduce the amount of time it took for the program to finish. The most dramatic jump in performance came between the 2 and 4-core runs. Using 4 cores, the program sped up by 1.58 times, with the BFS graph exploration being sped up about 1.90 times. However, after that, the run time plateaus. We would think this is the case as the overhead for managing the .

# Cores	Total Time	Total Speedup	Approximate BFS time	Approximate BFS Speedup
1	90.61	1.00	70.61	1.00
2	82.76	1.09	62.76	1.13
4	57.25	1.58	37.25	1.90
6	57.68	1.57	37.68	1.87
8	58.65	1.54	38.65	1.83

5. Conclusion

Overall we were able to speed up our program using parallelization, but not as much as we had originally hoped for. We achieved the highest speedup of 1.58x (~1.90x on BFS part) on four cores and found that speedup decreased with additional cores. A significant portion of our execution time went to reading our massive data file which we were not able to parallelize and which ultimately became a huge bottleneck. However, we are proud we were able to implement the game we originally set out to accomplish and both feel that we learned about parallelization in the process.

6. Code

Main.hs

```
module Main (main) where

import Lib
import GraphConstruction
import System.Exit ( die )
import System.Environment ( getArgs )

main :: IO ()
main = do
 [filenamePrefix, start, end] <- getArgsOrDie
 graph <- readDataParallel filenamePrefix
 runBFS graph start end

getArgsOrDie :: IO [String]
getArgsOrDie = do
 args <- getArgs
 if length args == 3 then return args
 else die "Usage: stack run <graph-filename-prefix> <from-vertex> <to-vertex>"
```

Lib.hs

```
module Lib
   ( runBFS
   ) where

import Data.List as List ( map )
import Data.Map as Map ( Map, insert, findWithDefault, singleton, toList )
import Data.Set as Set ( Set, null, toList, empty, difference, unions, union,
singleton )
import Control.Parallel.Strategies( using, rdeepseq, parBuffer, parList )
import Control.DeepSeq ( force )
import Data.List.Split ( chunksOf )

runBFS :: Map String (Set String) -> String -> String -> IO ()
runBFS neighborMap startWord endWord = do
 distances <- explore 0 (Set.singleton startWord) neighborMap (Map.singleton startWord
0) Set.empty
 let maxDegree = maximum $ List.map snd (Map.toList (force distances))
 -- Divide results by two since we should not count movie vertices
 putStrLn ("Distance from " ++ startWord ++ " to " ++ endWord ++ ": " ++ show
(toInteger (Map.findWithDefault (-1) endWord distances `div` 2) ))
 putStrLn ("Max distance: " ++ show (toInteger (maxDegree `div` 2)))

explore :: Int -> Set String -> Map String (Set String) -> Map String Int-> Set String
-> IO (Map String Int)
explore level vertices neighborMap distances visited
   | Set.null vertices = return distances
   | otherwise = do
       let newVisited = Set.union visited vertices
       next <- getNeighborsForSet vertices neighborMap visited
       let newDistances = foldl (\d vertex -> insert vertex (level + 1) d) distances
next
       explore (level + 1) next neighborMap newDistances newVisited

getNeighborsForSet :: Set String -> Map String (Set String) -> Set String -> IO (Set
String)
getNeighborsForSet vertices neighborMap visited = do
   let vertexList = Set.toList vertices
   let neighborSets = List.map (getNeighbors neighborMap visited) vertexList `using` 
parBuffer 7500 rdeepseq
   let chunkList = chunksOf 64 neighborSets
   let unionList = List.map Set.unions chunkList `using` parBuffer 8 rdeepseq
   return (Set.unions unionList)

getNeighbors :: Map String (Set String) -> Set String -> String -> Set String
getNeighbors neighborMap visited vertex = Set.difference (Map.findWithDefault 
Set.empty vertex neighborMap) visited
```

graphConstruction.hs

```
module GraphConstruction (parseData, buildGraph, readDataParallel) where

import Data.Set as Set ( Set, fromList, singleton, union )
import Data.Map as Map ( Map, empty, insert, insertWith, unionWith )
import Data.List.Split ( splitOn )
import Control.Parallel.Strategies( runEval, rpar, rseq )

parseData :: String -> IO (Map.Map String (Set String))
parseData filename = do
   contents <- readFile filename
   return (buildGraph (lines contents) Map.empty)

buildGraph :: [String] -> Map.Map String (Set String) -> Map.Map String (Set String)
buildGraph [] graph = graph
buildGraph (x:xs) graph = buildGraph xs graphWMovies
   where graphWMovies = foldr (\movie g -> Map.insertWith Set.union movie 
(Set.singleton name) g) graphWActor movies
         graphWActor = Map.insert name (Set.fromList movies) graph -- insert actor 
keys
         movies = splitOn "," movieStr
         [name, movieStr] = splitOn "\t" x

readDataParallel :: String -> IO (Map.Map String (Set String))
readDataParallel filenamePrefix = runEval $ do
   d1 <- rpar (parseData $ filenamePrefix ++ "1.csv")
   d2 <- rpar (parseData $ filenamePrefix ++ "2.csv")
   d3 <- rpar (parseData $ filenamePrefix ++ "3.csv")
   d4 <- rpar (parseData $ filenamePrefix ++ "4.csv")
   _ <- rseq d1
   _ <- rseq d2
   _ <- rseq d3
   _ <- rseq d4
   m1 <- rpar (Map.unionWith Set.union <$>  d1  <*>  d2)
   m2 <- rpar (Map.unionWith Set.union <$>  d3  <*>  d4)
   _ <- rseq m1
   _ <- rseq m2
   return (Map.unionWith Set.union <$>  m1  <*>  m2)
```