Assignment 2 report

Jesper Wingren
Jw223rn@student.lnu.se
1dt909

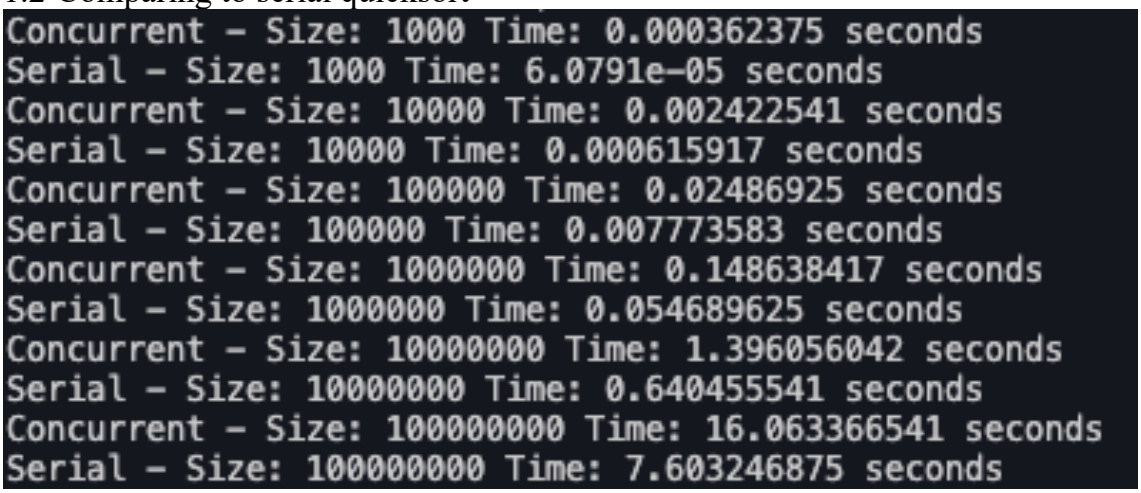
Innehållsförteckning

1. Concurrent quicksort

1.1 Setup

The setup of my concurrent quicksort is that it uses goroutines to concurrently sort the left and right part of my list. Using waitgroups, it makes sure the previous sorts are done before going on to the next step. The _sort method has a defer to the waitgroup in my struct making it call done only when the whole list is sorted. It then sets up the pivot while starting go routines for both sides of the list. The partition works as it does in a serial quicksort where the pivot is the last value in the list. All of this concludes to the quicksort algorithm to concurrently solve each side of the pivot value in the list.

1.2 Comparing to serial quicksort



Here are the times for my concurrent and serial quicksort. This is only run once with a randomized list thus meaning the times can differ a bit but gives us a good understanding of the times. Here we can see that the serial quicksort is faster than the concurrent which could be surprising but due to overhead and managing multiple threads it's not that surprising. The concurrent quicksort is best for special occasions with lots of data and on computers that can utilize threads optimally while the serial fits most cases. Both the concurrent and the serial quicksort uses the last item as pivot value and the speeds could therefore be quicker if I used for example median of three instead.

2. Concurrent Floyd-Warshall and Dijkstra

2.1 Setup

To introduce concurrency to the Floyd-Warshall algorithm I added a go func to start a goroutine for each source vertex. So, for each “box” in the adjacency matrix it concurrently checks the shortest distance and adds it to the matrix.

2.2 Results

In the tables below we can see the results of the program. What you can see here is that neither the Floyd-Warshall algorithm or Dijkstra cares about how many edges there are, it scales with the number of vertices. If the vertex amount is the same and the edges changes the time is approximately the same. You can also see that the Floyd-Warshall algorithm is quicker when there are more edges than vertices but is significantly slower than Dijkstra when there are less edges than vertices.

	Dijkstra (ms)	**Floyd-Warshall(ms)**
100 edges 100 vertices	0,172	0,875
100 edges 1000 vertices	1,3	222
100 edges 5000 vertices	20,8	26200
100 edges 10000 vertices	174	232000

	Dijkstra (ms)	**Floyd-Warshall(ms)**
500 edges 100 vertices	3,64	1,18
500 edges 1000 vertices	1,26	232
500 edges 5000 vertices	22,3	27000
500 edges 10000 vertices	216	244000

	Dijkstra (ms)	**Floyd-Warshall(ms)**
1000 edges 100 vertices	6,4	1,88
1000 edges 1000 vertices	5,56	224
1000 edges 5000 vertices	22,4	27000
1000 edges 10000 vertices	165	244000

3. N-ary search

3.1 Setup

My N-ary search is tested through different amount of goroutines and different sizes for the same number of workers. The time is calculated with an average of 1000 random searches in the array.

3.2 Results

```
Size: 10000000
Amount of workers: 1
Avg time: 2.957342893000002  ms
Size: 10000000
Amount of workers: 4
Avg time: 0.9316521720000007  ms
Size: 10000000
Amount of workers: 8
Avg time: 0.692655034  ms
Size: 10000000
Amount of workers: 16
Avg time: 0.6175012089999994  ms
Size: 10000000
Amount of workers: 32
Avg time: 0.5852913379999999  ms
```

```
Size: 10000
Amount of workers: 32
Avg time: 0.03928962900000004  ms
Size: 100000
Amount of workers: 32
Avg time: 0.04677853499999996  ms
Size: 1000000
Amount of workers: 32
Avg time: 0.08527328699999984  ms
Size: 10000000
Amount of workers: 32
Avg time: 0.5852913379999999  ms
```

We will start to discuss how well it scales with workers where you can see a clear increase of speed already going from 1 to 4 workers and the amount it speeds up decreases the greater the number of workers. This is because the program can't handle more workers before it reaches its peak efficiency. When slowly increasing the size, we can clearly see a slow increase in time but after moving from 1 million to 10 million in size the time increases a lot more than before although it's hard to find the exact time complexity it is something like O(n/amountOfWorkers) which is hard to compare to the time complexity of O(log(n)) which is the binary search. This really depends on the case of the search which one is faster and for larger arrays the binary heap will be quicker as it doesn't scale as hard as the N-ary does when increasing the amount