CS 6023: Final Report
The minimum fleet problem: a parallel approach

Govind Sankar, Dhruv Chopra
EE16B109, EE16B107

November 2018

Abstract

The paper "Addressing the minimum fleet problem in on-demandurban mobility" by Vazifeh et. al.[1] addresses the problem of finding the minimum fleet size a taxi service provider requires to service all of its customers. Through this project, we aim to improve the performance of their algorithm by parallelizing it and implementing it on the GPU. Our efforts have been concentrated on implementing and improving an existing parallel algorithm for the Hopcroft-Karp maximum matching algorithm. [2]

1 Introduction

Taxis are a necessity in an urban scenario. With companies like Uber flourishing, there has been a resurgence of interest, scientific and otherwise, in the field. Easy access via requests through booking using a simple mobile application have led to this surge in the use of taxis. One key interest of taxi service providers lies in minimizing their fleet size while ensuring that there are no inconveniences to the customers, in the form of wait times. This task is very involved, as it is dependent on parameters such as total demand, compute time and resources, fleet and driver availability, fuel cost, tolerance for punctuality and wait time, and current traffic and weather conditions.

In their paper[1], Vazifeh et. al. assume an a posteriori view of the problem. They redefine the problem statement like so: "Given a dataset of taxi trips, with start and end locations, and start and end times, what is the minimum fleet required to service this?". In this project, we have aimed to implement the methods described by the authors on a GPU, and have attempted to attain a speedup in their algorithm vis-a-vis the CPU implementation. To this end, we rely on an existing parallel algorithm for Hopcroft-Karp[2]. We use a NVIDIA Tesla K40, and have written the code in CUDA.

2 Construction of the Graph

The data was sourced from the New York City Taxi and Limousine commission's webpage[3], the same source the authors in [1] have used. The data was collected over various taxi trips in and around the city of New York. With regards to the data, we're concerned only with 4 parameters, namely: start and end coordinates of the trip, and pick up and drop off times.

We construct our graph keeping each individual trip as a node. Consider any two nodes a and b. Let the their pickup times be P_a, P_b, their drop off times D_a, D_b, their trip start coordinates B_a, B_b, and their trip end coordinates E_a, E_b. We claim that a directed edge (a, b) exists if

1. Trip a ends before trip b starts, that is $D_a < P_b$.
2. It is possible to reach the starting location of a from the ending location of b within a certain time δ and within time $P_b - D_a$.

For the purpose of this project, we have used the displacement, viz. crow flight distance, between the end point of a, E_a and the start point of b, B_b rather than their actual distance on road. We calculate the transit time between E_a and B_b as

$$T = \frac{\text{displacement}(E_a, B_b)}{v} \tag{1}$$

where we assume $v = 40km/hr$ to be be the average speed of cars in city traffic. The other parameter we have is δ. It is a function of how much we are willing to let a taxi travel to connect two trips. If δ is very large, for example, then we can easily connect two trips very far apart. Such a scenario would be wasteful for the taxi service provider owing to inefficiencies in time, and petrol usage. A more detailed analysis is given in [1]. We take δ to be equal to 15 minutes while constructing the graph. Thus, edge (a, b) will exist if

$$T \leq \delta$$

and

$$D_a + T < P_b$$

Note that the directed graph so obtained will be acyclic because time is causal, ie. $D_b > P_b$ and $P_a < D_a$.

The algorithm relies on the key observation that the minimum fleet problem is an application of the minimum vertex disjoint path cover, where each path will correspond to the trips served by one car. An example decomposition of the network into a graph, and it's subsequent path cover is shown in figure 1. This will then give us the number of taxis required.

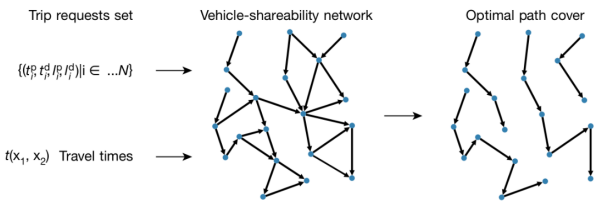


Figure 1: Representation of the data as a graph

3 Algorithm

The authors have described two algorithms: an offline version, where we assume we have full knowledge of the data for one day, and an online version where the data is received online. We will mostly be focusing on the offline version.

We have constructed the graph in Section 2. The next step of the algorithm is to find the path cover itself. For a general graph, this is an NP-Hard problem. However, one property of our graph is that it is a directed acyclic graph as described earlier. For DAGs, a polynomial time algorithm exists and is described below.

Algorithm 1: Minimum Fleet Problem

Data: `Trips` T, `Edges` E
Result: Number of taxis required N_T
1 Construct the directed graph G = (T,E)
2 Convert G into a bipartite graph $B = (T \cup T, E')$
3 Find the maximum matching M of B using the Hopcroft-Karp algorithm
4 $N_T \longleftarrow |T| - |M|$

The bipartite graph B has the trips T as both of its parts. We set the edge set E' of B as follows:

Undirected edge (u, v) exists in B if the directed edge $u \longrightarrow v$ exists in G.
Once we have the size of the maximum matching of B, which can then give us the size of the minimum path cover as

$$N_T = |T| - |M|$$

That is, the difference between the number of trips and the size of the matching gives us the number of taxis required. It remains to be explained *why* the maximum matching M in B gives us the vertex disjoint path cover when plugged into the above equation gives us the minimum path cover. An easy intuition for this is as follows. Initially we need one taxi for every node in T. But for every matched edge (u, v) in B, this represents a taxi that can be shared ie. the same taxi can serve trips u and v. The property of the matching also implies that this vertex will not get served by another taxi ie. the paths will be vertex disjoint. The formula thus follows.

The online version of the algorithm has a few incremental additions apart from this. They accumulate trip requests over a short period of time, say 15 minutes, and the offline version of the algorithm is then run on the collected trip requests.

4 Parallel algorithm

As is evident from the algorithm, the main computation of the algorithm lies in the Hopcroft-Karp step. Hopcroft-Karp is a promising candidate for parallelization because it is inherently parallel. As expected, there are already papers regarding the same. We reference once such paper here [2].

There are three major steps in Hopcroft-Karp algorithm.

1. Construction of the BFS tree from all free vertices.
2. Finding augmenting paths.
3. Augmenting the matching.

BFS has only some level of parallelism, owing to varied number of edges every node has. The promising factor here is that BFS is done from multiple nodes at a time. This is what the authors of [2] have done. The algorithm they have developed follow this pattern. Nodes are handled by threads one at a time, and the thread goes through all the edges of that node before moving on to the next.

The second step involves backtracking along the augmenting paths found in the first step to find vertex disjoint augmenting paths. Here, the only degree of parallelization exploitable is parallelizing across different found augmenting paths and validating whether they are vertex disjoint or not. The path has to be traced back sequentially and hence there is no parallelism possible in that regard.

The third step is augmenting the graph, which again has to be done sequentially.

5 Performance optimization

For optimizing performance, we first explored implementing the algorithm on multiple GPUs. However, for this to work, we need the algorithm to have independent instances executing together. Finding a maximum matching is a progressive and incremental problem as we need to find augmenting paths and augment them to the matching. Thus, there are no independent sections of the algorithm which could be implemented and executed in parallel on multiple GPUs. Given the need for synchronization of all data after every iteration, the overheads would outweigh the performance increase.

The main advantage that streams offer is hiding latency owing to data transfer and offering more workload to the GPU, so that it is saturated. This is not the case here because the data has to be completely transferred before the GPU can start. This is because of the previous argument: there is no independent workload. GPU saturation is also not an issue because we're using blocks of around 32 threads for the most part. Given more than 512 nodes, we have more than 16 blocks, which is the maximum number of active blocks possible.

With regards to shared memory, since there is no uniform data decomposition the only option is to store the whole graph in shared memory. This is not an option because the graphs we dealt with were typically of the order of hundreds of thousands of edges.

An optimization that is feasible is dynamic parallelism. The technique is ideal for non-uniform workloads. This was applied to the BFS kernel as every node has a different number of edges. With dynamic parallelism, vertices explores their neighbours in parallel instead of doing it sequentially. Every thread in the parent kernel (which corresponds to a node) call the child kernel with threads equal to the number of neighbours the given vertex has. So, the kernels are called as and when required with a variable number of threads.

In practice, this resulted in a higher execution time owing to overheads. As mentioned before, the other steps of the Hopcroft-Karp algorithm are inherently sequential and hence, no further parallelism is possible. Hence, we cannot apply dynamic parallelism elsewhere in the algorithm.

	CPU	**GPU**	**GPU with Dynamic Parallelism**
Sparse Taxi dataset	2.18465	3.7624032	47.7543117
Taxi dataset	34.6555	2.56796	343.12831575
Constructed dense graph	138.8285	305.71079855	886.00852965

Table 1: Raw data for all 3 datasets

6 Results

The numerical results are given in table 1.

The first thing we noted was that it was very difficult to obtain a speedup. This is shown in figures 2 and 3. The taxi dataset is highly non-uniform that makes it unsuitable for use with GPUs. We found that the CPU outperformed the GPU almost always. To contrast with the taxi dataset, we constructed a uniform graph to verify this hypothesis. Construction was such that the i^{th} node was connected to all nodes from $i + 1$ to 999. This gives an average degree close to that of the taxi dataset. The fact that uniformity gives a speedup is clear here; there is speedup of roughly 15 times over the sequential code. Moreover, the GPU performs takes a smaller time to operate on this dataset even compared to the sparse taxi dataset even though the number of edges is 10 times larger, which is extremely surprising.

In regards to block dimensions, 32 proved to be the most optimal. This is understandable is a number lower than this would not achieve full occupancy whereas a number higher than this would suffer from memory shortages (owing to the large number of edges) and the higher possibility of having two nodes of highly contrasting degrees.

Dynamic parallelism, while viable, still gives a slowdown. We suspect this could be because of multiple reasons

1. Launch and execution overheads

   This is likely, because one kernel is going to be launched per node, leading to 6511 kernels in the case of the taxi dataset. These will then launch kernels of their own. Moreover, these will all have extremely non-uniform number of threads assigned per kernel.

2. Small number of threads per kernel

   This is highlighted in the graphs. We see that as the nodes and average degree increases, the difference in time between the CPU and the GPU with dynamic parallelism decrease.

3. Memory inconsistencies over child kernels

   Since memory is only weakly consistent while the child kernels are running, the possibilities of race conditions increase. This requires us to do additional iterations to fix these inconsistencies. This is because the algorithm doesn't try to avoid race conditions using atomics, rather it takes an approach of fixing them as and when discovered.

7 Future Work

We have identified several possibilities of improving the performance of the code.

The first and foremost one is to make the dynamic parallelism efficient. The idea behind it is solid, and dynamic parallelism is suitable for graph algorithms. Some work has already been done on the same, for example, in [4].

The second avenue would be sorting the nodes by their degree beforehand. This should be done as a preprocessing step. We suspect this is the reason why the GPU performed extremely well in the constructed dense graph, as its vertices had degrees from 998 to 1 in order.

Another feature that could make the algorithm more work-efficient could be to not reconsider already matched vertices, in a process similar to compaction in parallel reduce. One upside of this work-efficiency could be that other blocks with work could take over, giving an improvement in performance. Lastly, since shared memory has not been used here at all, it is one other avenue that could be further explored.

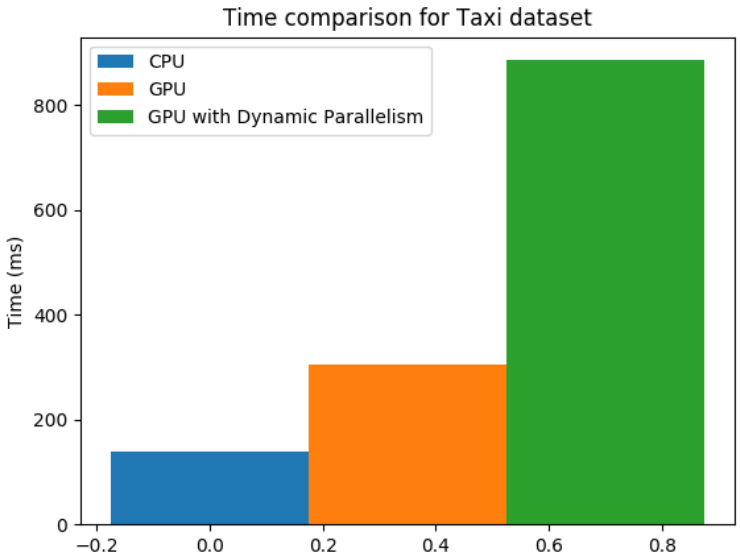
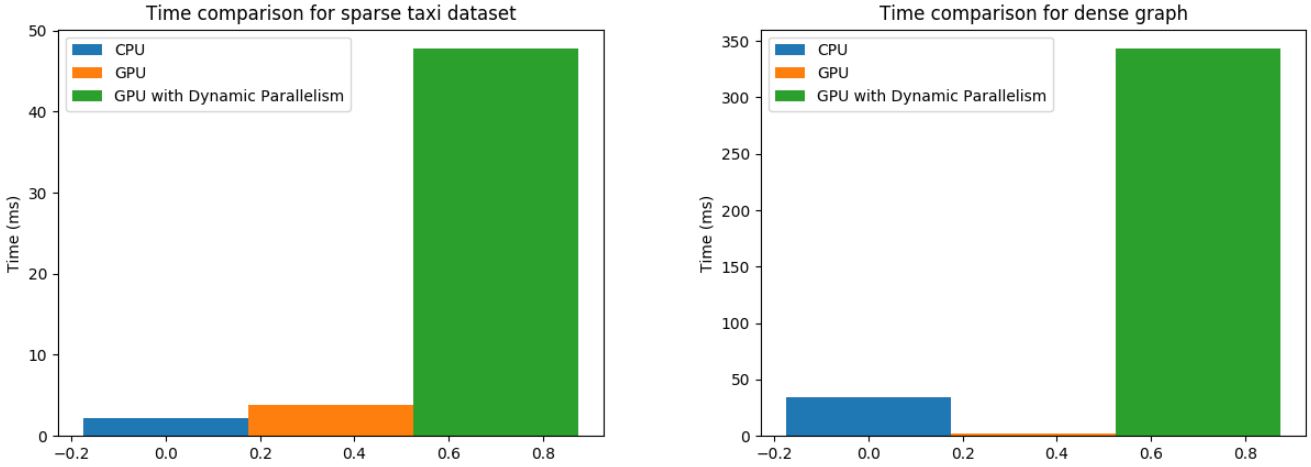


Figure 2: The full taxi dataset. It has 6511 nodes and 412750 edges and δ= 15 minutes.



(a) Taxi Dataset

(b) Dense graph

Figure 3: Comparison of performance in two different datasets. The sparse taxi dataset has 6511 nodes and 49888 edges, giving an average degree of 7.6. It corresponds to a very small value of δ (˜2 minutes). The dense graph was generated by us, with 999 nodes and 999*500 = 499500 edges, giving an average degree of 500.

References

[1] M. M. Vazifeh, P. Santi, G. Resta, S. H. Strogatz, and C. Ratti, "Addressing the minimum fleet problem in on-demand urban mobility," *Nature*, 2018.

[2] M. Deveci, K. Kaya, B. Ucar, and U. V. Catalyürek, "GPU accelerated maximum cardinality matching algorithms for bipartite graphs," *Euro-Par 2013*, 2013.

[3] "NYC taxi and limousine commission trip record data." http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml. Accessed: 2018-11-01.

[4] P. Zhang, E. Holk, J. Matty, S. Misurda, M. Zalewski, J. Chu, S. McMillan, and A. Lumsdaine, "Dynamic parallelism for simple and efficient GPU graph algorithms," *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms IA3*, 2015.