Computer Science

Capstone Report - Fall 2022

Parallel Computing for Solving Path-Based Traffic Assignment Problems

Harry Lee

supervised by
Olivier Marin, Zhibin Chen

Preface

This study investigates the solutions to the traffic assignment problem in a parallel computing context. The traffic assignment problem is one of the most significant problems in the field of traffic planning; such problems play a remarkable role in people's daily lives. In this paper, we are going to review the existing traffic assignment problem solutions, parallel computing fundamentals, and some earlier literature on its parallel solutions. The performance of a Spark implementation of our parallel solution will next be shown off. Finally, based on our findings, we will sketch some conclusions about this topic and cast some doubt on previous research.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Prof. Olivier Marin and Prof. Zhibin Chen for their guidance and assistance over the past semester. Being under their supervision is an honor, especially considering that we only ever communicated online. I would like to express my foremost appreciation to Prof. Marin for introducing me to this fascinating subject and inspiring me to conquer all the obstacles I would never have tried to confront without him. Additionally, I would like to show my gratitude to Prof. Zhibin Chen, who not only offers me the transportation and mathematical knowledge I require, but also invaluable assistance and guidelines that enable me to complete the entire project. I would also express my gratitude to Prof. Xianbin Gu for his earlier help on my computer science senior project, to Prof. Lihua Xu for her unwavering support throughout my undergraduate studies, and to all the computer science professors at NYU Shanghai for their guidance and care throughout my college years. Finally, I would like to thank my parents and friends for their love and support, whom without this would have not been possible.

Abstract

The traffic assignment or the transportation network equilibrium (NE) problem is defined to be finding the link flows given the origin-destination trip rates, the network, and the link performance functions[1]. Though these problems can be formulated as a convex program with linear constraints, the process of path enumeration is time-consuming. A set of algorithms has been proposed to avoid or alleviate the path enumeration problem [1, 2]. However, as the master problem remains highly computationally complex, we saw the need to distribute the original problem into a paralleled context[3]. This paper reviews several key areas within parallel computing for solving path-based traffic assignment problems, including problem definition, existing algorithms, and relevant parallel computing knowledge background; then presents and analyzes our Spark parallel implementation; in the end, it concludes with some refinements and questions on existing literature.

Keywords

parallel computing, distributed system, traffic assignment, transportation network equilibrium, NE, traffic planning, capstone, computer science, NYU, NYU Shanghai

Contents

1. Introduction

Traffic assignment is a fundamental tool for estimating and managing traffic flow in a transportation network. The *transportation equilibrium problem* (the *NE problem*), is one of the most recognized theories for traffic assignment. It is defined to be finding the link flows given the origin-destination trip rates, the network, and the link performance functions[1]. The goal of solving the NE problem is to find the minimum overall travel time for all origin-destination pairs. Though these problems can be formulated as a convex program with linear constraints, the process of path enumeration is time-consuming. A set of algorithms has been proposed to avoid or alleviate the path enumeration problem [1, 2]. However, as the master problem remains highly computationally complex, we saw the need to distribute the original problem into a paralleled context[3].

Parallel computing is a type of computation in which many calculations or processes are carried out simultaneously[4]. Large problems can often be divided into smaller ones, which can then be solved at the same time. To facilitate the design and implementation of parallel algorithms, several parallel computing frameworks are proposed, including MPI[5], Spark[6], and CUDA[7]. Meanwhile, programming models such as MapReduce have won increasing popularity for their easy-to-use and easily expressible properties, as well as their scalability over a large cluster of machines[8].

This paper is going to review the cross-discipline area between the NE problem and parallel computing, then present and analyze our Spark implementation. In particular, we are going to mathematically define the NE problem, reviewing its existing serial solutions as well as some applied parallel techniques. Then, we will dive into the area of parallel computing, reviewing several key concepts as well as commonly used tools and frameworks. With all knowledge set up, we will introduce our parallel implementation with Apache Spark and analyze its performance. Finally, we will highlight some limitations and improvements that could be taken to our improvement, and bring some questions to existing literature.

2. Related Work

2.1. The Traffic Assignment Problem

2.1.1. Problem Definition

The basic problem of the traffic assignment or the transportation network equilibrium problem is to find the link flows given the origin-destination trip rates, the network, and the link performance functions. It is based on the behavioral assumption that each motorist travels on the path that minimizes the travel time from origin to destination.[1] The above statemate can be mathematically expressed as a convex optimization problem, in which we find the argmin for the objective function:

$$\min z(\mathbf{x}) = \sum_a \int_0^{x_a} t_a(\omega) d\omega \tag{1}$$

subject to

$$\sum_k f_k^{rs} = q_{rs} \ \ \forall\, r, s \tag{2}$$

$$f_k^{rs} \geq 0 \ \ \forall\, k, r, s \tag{3}$$

And with the definitional constraints

$$x_a = \sum_r \sum_s \sum_k f_k^{rs} \delta_{a,k}^{rs} \ \forall a \tag{4}$$

In this formulation, $z(\mathbf{x})$ is the objective function that is being minimized; x_a is the flow on arc a, and we define $\mathbf{x} = (..., x_a, ...)$; t_a is the travel time function on arc a, we define t_a to be $t_a(x_a)$; f_k^{rs} is the flow on path k connecting O-D pair r-s; q_{rs} is the trip rate between origin r and destination s; $\delta_{a,k}^{rs}$ is an indicator variable, $\delta_{a,k}^{rs} = 1$ if link a is on path k between O-D pair r-s and $\delta_{a,k}^{rs} = 0$ otherwise.

2.1.2. Proposed Solutions

There are several algorithms to find the solution to the optimization problem. The *simplex algorithm* is the one that is proposed most early and is used widely among most commercial solvers[9]. However, the procedures of this algorithm result in path enumeration which in turn produces a huge whole matrix, which can be hardly distributed among a cluster.

To avoid path enumeration, the *column generation algorithm*[2] presents an iterative solution procedure. As illustrated in figure 1, the procedure convents the original master problem (MP) into a restricted master problem (RMP); in other words, it first solves a simpler NE problem with fewer paths. Then, a set of sub-problems (SP) are tested, determining whether the solution to the RMP also satisfies the solution of the MP. If not, the SP is added to the RMP, and we solve the RMP in the new iteration; otherwise, the iteration stops and we get the final result. Column generation guarantees that we can get an optimal solution in the end, but we could always choose to stop at any iteration, then an approximated result will be generated.

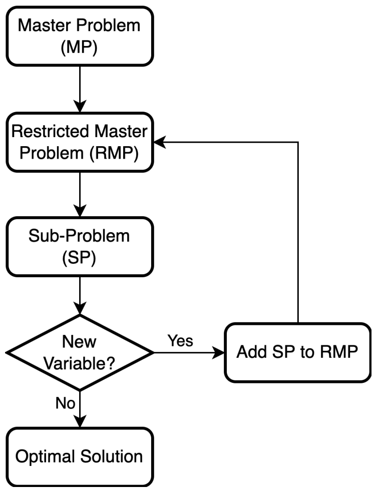


Figure 1: Column generation algorithm workflow

Similar to the column generation algorithm, the *Frank-Wolfe algorithm* (*linear approximation method*) can also avoid path enumeration[10]. The basic idea behind this algorithm is gradient descent, in which we find the shortest route for each iteration and move a descendant step toward the optimal solution. The most time-consuming part of this algorithm is finding the shortest path, while a lot of algorithms have been proposed to complete this task, including the PARTAN algorithm (and its variants), simplicial decomposition, successive quadratic approximations, the TDLTP algorithm, the DOT algorithm, and the CHRONOSPT algorithm, while several studies show that these algorithms are able to achieve a certain level of parallelization[10, 11].

Although the Frank-Wolfe algorithm is widely applied, [3] suggests that its computational efficiency is not optimal. It argues that the *gradient projection algorithm* can outperform the origin-based algorithms in large-scale transportation networks, and this algorithm could also be solved in parallel. The basic idea of the gradient projection algorithm is approximating the objective function (equation 1) by the second-order Taylor expansion. Then the original integral and summation can be separable with respect to O-D pairs, which can then be solved in parallel.

2.1.3. Parallel Implementations

Several studies have been investigating the parallelization of the NE problem. [12, 13] propose methods to parallelize the column generation method, [10, 3] propose methods to parallelize the Frank-Wolfe algorithm, [3] proposes methods for parallelizing the gradient projection algorithm, while [10, 11] focus on a smaller area - *temporal shortest paths*, which is a sub-step of the Frank-Wolfe algorithm. These studies lack a common evaluation on the same dataset, so it is hard to compare their efficiency directly. However, they draw several common conclusions:

1. Multi-thread programming with shared memory performs much better than massage passing through the network.
2. The best algorithm for the serial environments is likely to be the worst in a parallel context.
3. Network decomposition and network replication are 2 ways to divide the master problem into sub-problems, as the former often results in a lot of communication among processors while network bandwidth is considered to be a scarce resource, all studies choose the latter approach.

2.2. Distributed Systems and Parallel Computing

2.2.1. Parallel/Distributed Frameworks

Parallel computing is a type of computation in which many calculations or processes are carried out simultaneously[4]. Large problems can often be divided into smaller ones, which can then be solved at the same time. To facilitate the design and implementation of parallel algorithms, several parallel computing frameworks are proposed, including MPI/OpenMP[5], Spark[6], and CUDA[7]. Among these frameworks, MPI/OpenMP and Spark are suitable for general parallel computing on clusters with shared memory or network, while CUDA is designed for parallel computing over GPU. As we are mainly interested in a general framework in which we can apply our parallel solution with commercial linear solvers, Spark and MPI come to be our preferred choices.

Apache Spark is an open-source unified analytic engine for large-scale data processing from UC Berkley that exploits in-memory computation for solving iterative algorithms and can be run in traditional clusters such as Handoop[14]. The major advantages of Spark are its convenient usage of MapReduce programming model (section 2.2.2), the easy adaptation from serial implementation to a parallel context, and efficient fault tolerance support[3]. OpenMP/MPI (message passing interface), on the other hand, provides users with more flexibility. OpenMP/MPI provides a solution mostly oriented to high-performance computing but susceptible to faults.

[14] conducts a quantitative comparison between Spark and MPI/OpenMP, their conclusions show that though Spark has much less computational efficiency compared to MPI/OpenMP, it is preferred in several situations:

- with need for a distributed file system with failure and data replication management
- with need for elastic scaling at runtime
- with need for a set of data analysis and management tools

2.2.2. The MapReduce Model

MapReduce is a programming model and an associated implementation for processing and generating large data sets proposed by Jeffrey Dean and Sanjay Ghemawat at Google[8]. The idea of MapReduce was formulated around the year 2003 and has gained increasing popularity in the area of distributed systems due to its easy-to-use, easily expressible, and large-cluster-friendly properties. The structure of MapReduce has highlighted several key factors in the design of a distributed system. Basically, users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key.

When MapReduce executes, the master worker first split the input data into M pieces and assigns M map tasks and R reduce tasks to the slave workers. In the map phrase, the map workers read the input split in the format of (k, v) pairs and generate intermediate output of (k, v) pairs, then store the output in their local memory buffer. Periodically, the map workers write data from local memory to the local disk partitioned into R regions, and pass the location to the master. When the master receives messages from the map worker, it notifies the reduce workers to read data from the map workers and append the output file to the reduce function. When all map and reduce workers end, the master program returns.

The MapReduce model hides the parallelization details including fault tolerance, locality optimization, and load balancing. For fault tolerance, the slave workers are pinged periodically while the failure machines are marked. Completed map tasks on the failure machines are re-executed because they contain output files needed for the reduce functions. However, completed reduce tasks are not re-executed because their outputs are stored in the global storage. For locality optimization, MapReduce assigns map tasks according to the partition of the input data each map worker owns, which saves bandwidth, as bandwidth is considered a scarce resource in the context of distributed systems. When the MapReduce function is nearly finished, ongoing tasks will be repeatedly run on other idle machines, which helps to accelerate execution.

2.2.3. Evaluation on Parallel Algorithms

Evaluations of NE problem parallel solvers are often conducted over data sets of the city transportation network with a different range of sizes, either on a cluster with shared memory, or clusters in which machines are connected by a network. During the evaluation, several variables are considered, including the NE algorithm, number of nodes (machines), network bandwidth, and network shared memory [10, 11, 3]. Several evaluation metrics are defined as follows:

- Execution time, including the minimum, maximum, and average time.
- Speedup. Speedup is defined to be

  $$a(\omega, p) = \frac{T_s(\omega)}{T(\omega, p)}$$

  , where $T_s(\omega)$ is the computing time of the serial algorithm of size ω, and $T(\omega, p)$ is the computing time of the parallel algorithm of size ω on p processors.
- Relative burden. The relative burden is defined to be

  $$b(\omega, p) = \frac{T(\omega, p)}{T_s(\omega)} - \frac{1}{p}$$

  . The advantage of using the relative burden is that it can provide an estimate of the "best" speedup[10].
- Efficiency of processors. The efficiency of processors is defined to be

  $$E(\omega, p) = \frac{a(\omega, p)}{p}$$

  , which gives us information on how efficiently each processor works. Due to communication overhead, $E(\omega, p) < 1$ for $p > 1$[3].

3. Solution

Both the Frank-Wolfe algorithm and column generation algorithm avoid the problem of path enumeration, so we take these 2 algorithms as our experiment subjects. We believe though the gradient projection algorithm gains adequate performance, its structure shares many similarities with the Frank-Wolfe algorithm, thus we have ignored it in our solution design.

3.1. Serial Implementation and Unit Tests

Algorithm 1 Frank-Wolfe Algorithm

```
Require: $network$, $tripRtFunc$, $odPs$
    ▷ Step 0: find a feasible solution (using shortest path)
 1: $shortestPaths \leftarrow []$
 2: for $odP$ in $odPs$ do
 3:     $shortestPath \leftarrow dijkstra(network, tripRtFunc, curTfc, odP)$
 4:     add $shortestPath$ to $shortestPaths$
 5: end for
 6: $curTfc, z \leftarrow assignTraffic(network, shortestPaths, curTfc)$
 7: while $true$ do
    ▷ Step 1: solution of linearized subproblem
 8:     $shortestPaths \leftarrow []$
 9:     for $odP$ in $odPs$ do
10:         $shortestPath \leftarrow dijkstra(network, tripRtFunc, curTfc, odP)$
11:         add $shortestPath$ to $shortestPaths$
12:     end for
13:     $newTfc, \mathbf{newZ} \leftarrow assignTraffic(network, shortestPaths, curTfc)$
    ▷ Step 2: find optimal step size (we can use the $ParTan$ method, below is also one way)
14:     $stepSize = 0.5$
15:     for $i \leftarrow range(LOOP)$ do
16:         $gradient \leftarrow calculateGradient(network, odPs, curTfc, newTfc, stepSize)$
17:         if $gradient < 0$ then
18:             $stepSize = stepSize + \frac{1}{2^{i+1}}$
19:         else
20:             $stepSize = stepSize - \frac{1}{2^{i+1}}$
21:         end if
22:     end for
    ▷ Step 3: Update current traffic
23:     $curTfc \leftarrow curTfc + stepSize \times (newTfc - curTfc)$
    ▷ Step 4: Check convergence
24:     $newZ = calculateZ(network, odPs, curTfc)$
25:     if $z - newZ < \varepsilon$ then
26:         break
27:     end if
28: end while
29: return $curTfc, z$
```

Algorithm 1 shows the details of the Frank-Wolfe algorithm and algorithm 2 shows our implementation of the column generation algorithm.

We use Java as our programming language for its convenience in packaging, testing, and connection to parallel computing frameworks such as Spark and MPI, as well as the balance between development efficiency and run-time efficiency provided by this programming language. In our serial implementation for both algorithms, we utilize Dijkstra's algorithm to find the shortest path. Dijkstra's algorithm is itself a serial algorithm that can hardly be decomposed when finding the shortest path between a given OD pair. However, the advantage of such a method is that besides low time and space complexity, given an orientation node, we can find the shortest path to all other nodes in one go, without further overhead. As such, Dijkstra's algorithm comes to be our first choice of implementation. In our implementation, we cache the result of Dijkstra's algorithm for each orientation node and reuse it when possible.

Algorithm 2 Column Generation

```
Require: $network$, $tripRtFunc$, $odPs$
    ▷ Step 0: find a feasible solution (using shortest path)
 1: $shortestPaths \leftarrow []$
 2: for $odP$ in $odPs$ do
 3:     $shortestPath \leftarrow dijkstra(network, tripRtFunc, curTfc, odP)$
 4:     add $shortestPath$ to $shortestPaths$
 5: end for
 6: $maskedNetwork \leftarrow maskNetwork(network, shortestPaths)$
 7: while $true$ do
    ▷ Step 1: solve master problem
 8:     $curTfc, z \leftarrow frankWolfe(maskedNetwork, odPs)$
    ▷ Step 2: Solve subproblem
 9:     $newPathFound \leftarrow false$
10:     for $odP$ in $odPs$ do
11:         $shortestPath, z \leftarrow dijkstra(network, tripRtFunc, curTfc, odP)$
12:         if $prevTravelTime > curTravelTime$ then
13:             $newPathFound \leftarrow true$
14:             add $shortestPath$ to $shortestPaths$
15:             $maskedNetwork \leftarrow maskNetwork(network, shortestPaths)$
16:         end if
17:     end for
18:     if $\neg newPathFound$ then
19:         break
20:     end if
21: end while
22: return $curTfc, z$
```

In our implementation for step 2 of the Frank-Wolfe Algorithm (as shown in algorithm 1, the part for finding optimal step size), instead of taking the state-of-the-art methods which claim to have a quicker convergence rate, we simply take the bisection method - by measuring the direction of the gradient at the central point of the search area in each iteration, we narrow down the size of the search area to half - which not only guarantees a fixed time complexity but also gives us an accurate result as the precision increases exponentially with each iteration. Furthermore, the fixed precision and time complexity define a clear goal for us to design the parallel solution, making the comparison between parallel and serial solutions more convincing.

Our implementation of the column generation algorithm takes advantage of the Frank-Wolfe implementation to solve its subproblem (step 2 in algorithm 2).

As for quality insurance, we write unit tests with JUnit[15] for each class and key function. This makes sure we will always have accurate results and all components from both serial and parallel implementations produce the same outcome.

3.2. Parallel Implementation with Apache Spark

We decided to use Apache Spark[6] over MPI[5] as our parallel computing framework. Compared to MPI, Apache Spark provides a more accessible interface for controlling cluster configuration; the MapReduce model along with the RDD transformations and actions design patterns makes us easy to enforce a solution in a short time. We have 2 versions of implementation, section 3.2.1 presents a solution with larger granularity in which we divide tasks for each orientation in finding the shortest path (Parallel-A), and section 3.2.2 presents a solution with smaller granularity in which we further divide tasks in the calculation of objective function and optimal step size finding (Parallel-B).

3.2.1. Split by OD Pair (Parallel-A)

As shown in some literature[3, 10], dividing tasks for each orientation node in finding the shortest path is a preferred solution, which also utilized the characteristic of Dijkstra's algorithm. As a result, our implementation of finding the shortest path and completing the all-or-nothing traffic assignment includes the following steps:

- Step 0 (preparation before all iterations): broadcast orientation-destination pairs.
- Step 1: Calculate and broadcast the current trip rate (distance) graph given the trip rate function and current traffic.
- Step 2: Map phase: for each orientation, create a sub-task on a worker node: run Dijkstra's

algorithm, assign all traffic from this orientation on the shortest path found. Then return the $(key, value)$ pair where the key is the orientation index, and the value is the all-or-nothing traffic assignment in the form of a 2D matrix.

- Step 3: Reduce phase: for each worker node, add the corresponding entities of the 2D matrix together so their sum represents the whole new traffic.

3.2.2. Smaller Granularity: Split Objective Function (Parallel-B)

Some other literature suggests that decomposing the graph is also a choice[11], while Florian et al.[10] suggests that the calculation of the objective function and finding the optimal step size part (step 2 of algorithm 1) can be further distributed. As a result, on top of the Parallel-A version, we made some modifications so that the current traffic is changed into the format of Spark RDD. In other words, the current traffic presents as a distributed dataset that is automatically maintained by Spark - at the call of calculating the objective function, each worker node will calculate the sum of the objective function values for links that start from the origination node it is assigned to; finally, in the reduce phrase, all objective function values will be summed together.

3.3. Application Interface

For sake of the completeness of this project and the convenience of experiments, we used Maven to pack the project into a *jar* package, leaving the following entities as Java main function arguments: dataset name, mode (parallel/serial), and algorithm (Frank-Wolfe or Column Generation). The application interface is able to work with the *spark-submit* command on the NYU Shanghai HPC with Slurm Workload Manager, which makes it possible for large-scale experiments.

4. Results

In this section, we will introduce the dataset as well as the test environment we are working on, followed by our experiment results and analysis.

4.1. Experiment Setup

We conduct our experiment on the NYU Shanghai HPC cluster with Slurm Workload Manager. The cluster provides 43 nodes for parallel computing experiments with different models of Intel x86 CPUs. All processors within the cluster have a minimum number of 12 cores, with Intel Hyper-Threading Technology, a minimum of 24 threads for each node is guaranteed.

In our experiment, we are mainly interested in the following questions:

1. How is the performance difference between Frank-Wolfe algorithm and column generation?
2. As the number of worker nodes increases, how will the performance change?
3. How large is the difference between multi-node and multi-core configurations?
4. With respect to task granularity, is a larger granularity (Parallel-A) preferred or a smaller one (Parallel-B)?

We take advantage of the public transportation database for our experiment subjects. Their configurations are listed in Table 1. Networks such as Chicago Regional require more than 192 GB of memory for each node, and the computation costs nearly 1 hour for each experiment. Due to resource limits, we focus on networks with medium sizes such as Winnipeg and Barcelona. We control the following variables and repeat them 5 times for each setting of the experiment: number of nodes, number of cores per node, mode (parallel/serial), algorithm (column generation/Frank-Wolfe), and dataset (Winnipeg/Barcelona).

Network	**Zones**	**Nodes**	**Links**
Winnipeg	147	1052	2836
Barcelona	110	1020	2522
Chicago Regional	1790	12982	39018

Table 1: Test dataset network configurations

4.2. Evaluation

4.2.1. Parallel Computation Granularity

As illustrated in section 3.2, we have implemented 2 versions of parallel solution - the one with larger granularity and the one with smaller granularity. Before our comparison between parallel and serial solutions, our first investigation falls on selecting one from our two parallel implementations.

Table 2 shows our preliminary result for comparing Parallel-A and Parallel-B implementations. Given 8 nodes and 8 cores for each node, the overall execution time for Parallel-A is around half of the time for Parallel-B. Finding optimal step size (step 2 of Algorithm 1) is the place where the difference comes to be the largest. Our interpretation is that, compared to finding the shortest path, calculating the objective function for each link is much less time-consuming. Thus, the overhead of initializing a MapReduce function may worsen overall performance.

Implementation	**Algorithm**	**Network**	**Total Time**	**GetNewTfc**	**FindOptStep**
Parallel-A	FW	Winnipeg	35.51	27.68	5.35
Parallel-A	FW	Barcelona	31.95	23.56	5.07
Parallel-A	CG	Winnipeg	102.39	84.31	11.73
Parallel-A	CG	Barcelona	75.41	59.32	10.41
Parallel-B	FW	Winnipeg	68.52	40.84	23.81
Parallel-B	FW	Barcelona	55.36	34.26	16.08
Parallel-B	CG	Winnipeg	194.23	129.03	62.93
Parallel-B	CG	Barcelona	143.06	85.19	40.29

Table 2: A comparison between Parallel-A implementation and Parallel-B implementation with 8 nodes and 8 cores on each node in terms of their total execution time, time for getting new traffic, and time for finding optimal step size (unit: second). The result is the average of 5 experiments.

In our consequent experiments, we will abandon the Parallel-B solution and refer the Parallel-A implementation directly as the "parallel solution".

4.2.2. NE Algorithms

In this section, we compare the performance of our implementation of Frank-Wolfe algorithm and column generation. Table 3 shows the performance of the two algorithms averaged over our whole experiments. From a general perspective, column generation takes $2 \sim 3$ times for execution compared to Frank-Wolfe. As both algorithms avoid the path-enumeration problem, column generation shows nearly no advantage over Frank-Wolfe.

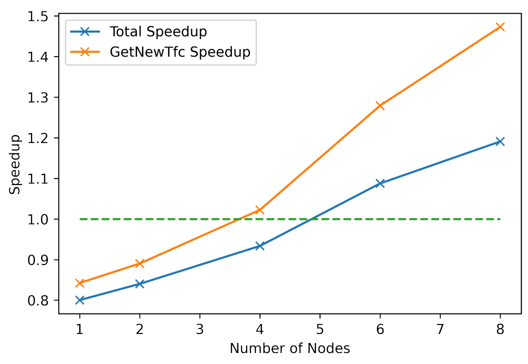
	Winnipeg	**Barcelona**
Frank-Wolfe	41.90	32.08
Column Generation	109.49	79.32

Table 3: A comparison between Frank-Wolfe algorithm and column generation in terms of their total execution time (unit: second), averaged over all our experiments.

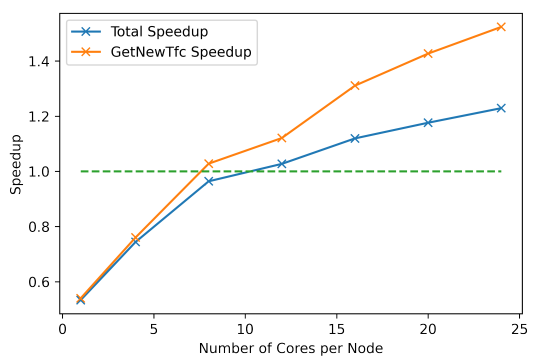
For a closer look, we count the time of the iteration in the main loop of column generation (the *while* loop in Algorithm 2) and keep track of how many links are involved in each iteration of the sub-problem. Results show that the times of iteration are usually below 5, while each sub-problem usually contains more than 80% of the links of the original graph. In conclusion, the effect of "reducing the original problem and remove unnecessary links" of column generation does not help with increasing algorithm performance, and it also performs poorly in decreasing space complexity.

4.2.3. Cluster Configuration

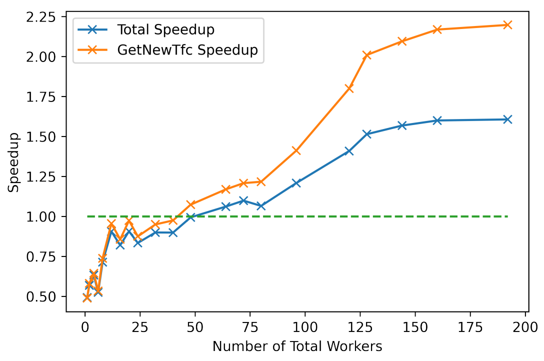
In this section, we will discuss the performance of the parallel algorithms in different cluster settings. Particularly, we are interested in how the number of nodes, the number of cores per node, and the total number of workers influence algorithm performance. Figure 2 shows the results corresponding to each variable in terms of line charts; appendix A shows one part of the data we use for the graphs. The green line in each graph represents our baseline - performance of the serial implementation.



(a) Average speedup with respect to the number of nodes



(b) Average speedup with respect to the number of cores per node



(c) Average speedup with respect to total number of workers ($numWorkers = numNodes \times numCoresPerNode$)

Figure 2: Algorithm performance with respect to settings of the cluster, results are averaged over our whole experiments.

Our result shows that, either a low amount of nodes or a low amount of cores per node may lead to a performance that is poorer than the serial solution. Generally, the increase in nodes and the number of cores per node will lead to an increase in performance. However, the increase in the number of workers alone may not lead to an increase in performance - the balance between nodes and cores should be considered. We believe that the effect of introducing a new node is more conspicuous than adding a new core, especially considering Intel Hyper-Threading Technology when the concept of *core* in Spark may be different from the concept of *core* in Intel CPU. Our data support this belief - to achieve an overall speedup of 1.2, we need approximately 8 nodes (2b), but approximately 22 cores (2a). In conclusion, only when the number of cores and nodes reach a threshold will parallel computing gain greater performance than the serial solution.

5. Discussion

5.1. Challenges

During the entire project, we've encountered 3 challenges: programming language, HPC cluster, and online communication.

Developing a Spark program requires in-depth knowledge of Java programming language, and associated toolchains such as JUnit and maven. Thus, learning the relevant technology stack and getting ready for building a reliable engineering project takes me a lot of time.

The NYU Shanghai HPC cluster uses Slurm as the work scheduler. Though it provides a Spark module where I can directly import, it did not provide instructions on how to launch a cluster project. Nor did the official Spark documentation mention the Slurm cluster. To make things work, I looked through a lot of other universities and clusters' handbooks including Stanford and Princeton, and finally figured out the ways for starting master and slave nodes manually, as well as ways for memory control in *spark-submit*. In addition, the configuration of the HPC makes some links between nodes unreachable, which increases the difficulties for my experiment.

The whole project was supervised by my 2 professors online throughout the whole semester. We have to overcome the unstable virtual environment and conquer the difficulties of being in different time zones.

5.2. Reflection on Related Works

The study of Chen et al.[3] shows that the speedup of parallel algorithms works better in a larger network. Due to hardware limitations on memory and CPU cores, we are not able to validate this argument. However, our result on graphs with 1/10 size of their test data still shows a conspicuous gap to their result. Reviewing their details, I am wondering if it is due to the difference in the data structure which we use to transfer messages between workers - their study uses an adjacency list, while our implementation uses an adjacency matrix. The advantage of our data structure is the convenience in the *reduce* phase for combining the traffic network. However, I then realize that the transportation network is more like a sparse matrix, and the use of an adjacency matrix increases a lot of space complexity. As a result, space complexity may be the reason for our gaps.

The study of Florian et al.[10] suggest that all steps in Frank-Wolfe algorithms could be decomposed and reach a higher performance. However, our results do not support so. The distributed calculation of the objective function needs a broadcast of the current traffic graph to all worker nodes, which is time-consuming. In addition, calculating the result of the objective function is not a computationally demanding task, thus the time saved can hardly exceed the communication overhead.

5.3. Limitations

There are mainly 2 limitations in our project - hardware resources and software structure.

The study of Chen et al.[3] utilizes more than 600 threads, while 192 is our maximum. In addition, we do not have enough memory quota to run experiments with large graphs such as Austin, Philadelphia, and Chicago, as such, it may be hard for us to compare their results directly. In the meantime, as our CPU cores are shared between users, it is possible our work efficiency is being influenced by other works.

Due to limitations in time and effort, we have not explored some details in the implementation of NE solutions, including but not limited to the data structure storing the network, methods for finding the optimal step size, and some parallel shortest path solutions[11]. These works may increase the performance of our existing solutions.

5.4. Future Work

Our results suggest the following work which may help boosting parallel computing performance:

1. Use adjacency list to represent network graph.
2. Get a deep understanding of JVM memory management, try decreasing the need for memory so we can test with a larger graph with more nodes.
3. Try paralleling the shortest path algorithms.

6. Conclusion

In this paper, we mathematically introduce and define the Traffic Assignment Problem along with existing serial and parallel solutions in the literature. We introduce our implementation of both serial and parallel versions of 2 NE algorithms and give a holistic assessment of it. In particular, our contributions are as follows:

- Review the cross-discipline area between the NE problem and parallel computing, including the definition of the NE problem, as well as relevant knowledge and toolchain for parallel computing.
- Implement serial and parallel solutions for Frank-Wolfe algorithm and column generation, discussing the implementation details.
- Conduct a holistic assessment of our implementations and compare it with existing studies, proposing directions for improvement and refinement. computing,

References

[1] Y. Sheffi, *Urban Transportation Networks: Equilibrium Analysis with mathematical programming methods.* Prentice-Hall, 1985.

[2] F. He, Y. Yin, and S. Lawphongpanich, "Network equilibrium models with battery electric vehicles," *Transportation Research Part B: Methodological*, vol. 67, pp. 306–319, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0191261514000915

[3] X. Chen, Z. Liu, and I. Kim, "A parallel computing framework for solving user equilibrium problem on computer clusters," *Transportmetrica A: Transport Science*, vol. 16, no. 3, p. 550–573, 2020.

[4] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing.* USA: Benjamin-Cummings Publishing Co., Inc., 1989.

[5] Intel, "Intel® mpi library." [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html

[6] Apache, "Apache spark™ - unified engine for large-scale data analytics." [Online]. Available: https://spark.apache.org/

[7] Nvidia, "Cuda toolkit - free tools and training," Sep 2022. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[8] J. Dean and S. Ghemawat, "Mapreduce," *Communications of the ACM*, vol. 51, no. 1, p. 107–113, 2008.

[9] K. G. Murty, *Linear programming. Murty.* John Wiley amp; Sons, 1976.

[10] M. Florian, I. Chabini, and Éric Le Saux, "Parallel and distributed computation of shortest routes and network equilibrium models," *IFAC Proceedings Volumes*, vol. 30, no. 8, pp. 1259–1264, 1997, 8th IFAC/IFIP/IFORS Symposium on Transportation Systems 1997 (TS '97), Chania, Greece, 16-18 June. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667017439942

[11] N. Tremblay and M. Florian, "Temporal shortest paths: Parallel computing implementations," *Parallel Computing*, vol. 27, no. 12, p. 1569–1609, 2001.

[12] S. Basso and A. Ceselli, "Distributed asynchronous column generation," *Computers amp; Operations Research*, vol. 146, p. 105894, 2022.

[13] M. Yu, V. Nagarajan, and S. Shen, "Improving column generation for vehicle routing problems via random coloring and parallelization," *INFORMS Journal on Computing*, vol. 34, no. 2, p. 953–973, 2022.

[14] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf," *Procedia Computer Science*, vol. 53, p. 121–130, 2015.

[15] T. J. Team, "The 5th major version of the programmer-friendly testing framework for java and the jvm." [Online]. Available: https://junit.org/junit5/

A. Experiment Results in Table

Appendix A shows part of the data that we use to group and generate tables and graphs. The whole dataset and program package can be found at GitHub (https://github.com/Harrilee/parallel-ne-solver).

Nodes	Cores/Node	Method	Time	NewTfcTime	TotalSpdup	NewTfcSpdUp
1	1	serial	36.86	31.28	1.00	1.00
1	1	parallel	74.71	63.88	0.49	0.49
1	4	parallel	52.02	43.23	0.71	0.72
1	8	parallel	39.95	31.87	0.92	0.98
1	12	parallel	40.67	32.71	0.91	0.96
1	16	parallel	44.51	35.47	0.83	0.88
1	20	parallel	40.63	32.11	0.91	0.97
1	24	parallel	44.10	35.06	0.84	0.89
2	1	parallel	64.77	53.83	0.57	0.58
2	4	parallel	52.90	44.48	0.70	0.70
2	8	parallel	41.69	33.79	0.88	0.93
2	12	parallel	41.10	33.12	0.90	0.94
2	16	parallel	38.03	30.13	0.97	1.04
2	20	parallel	41.00	32.07	0.90	0.98
2	24	parallel	38.14	29.32	0.97	1.07
4	1	parallel	66.17	55.21	0.56	0.57
4	4	parallel	49.22	40.94	0.75	0.76
4	8	parallel	39.45	31.54	0.93	0.99
4	12	parallel	37.80	30.00	0.98	1.04
4	16	parallel	33.96	25.86	1.09	1.21
4	20	parallel	34.58	25.69	1.07	1.22
4	24	parallel	31.55	22.88	1.17	1.37
6	1	parallel	70.18	58.61	0.53	0.53
6	4	parallel	47.88	39.57	0.77	0.79
6	8	parallel	35.42	28.11	1.04	1.11
6	12	parallel	33.55	25.88	1.10	1.21
6	16	parallel	30.71	22.10	1.20	1.42
6	20	parallel	26.16	17.38	1.41	1.80
6	24	parallel	23.51	14.93	1.57	2.10
8	1	parallel	70.52	59.02	0.52	0.53
8	4	parallel	46.33	38.09	0.80	0.82
8	8	parallel	35.51	27.68	1.04	1.13
8	12	parallel	29.29	21.51	1.26	1.45
8	16	parallel	24.34	15.56	1.51	2.01
8	20	parallel	23.04	14.42	1.60	2.17
8	24	parallel	22.95	14.23	1.61	2.20

Table 4: The entire results of Frank-Wolfe algorithm experiment on the Winnipeg dataset listed in table. All entries are the average of 5 experiments.