# Abstract

**Cab\_Share** is the implementation of a cab sharing and scheduling problem wherein a fixed number of cabs run between several locations in the city and the goal is to maximize the total revenue earned by the cabs, by serving as many requests as possible within a stipulated time frame. The initial position of all cabs and the schedule of all requests is known beforehand. The output gives us data about the requests served by the cabs and the total revenue earned. Floyd-Warshall algorithm for calculating shortest distances in a weighted graph is used to optimize the paths followed by the cabs to serve the requests.

**Parallelization** of the above mentioned implementation takes into account various constructs in the code that perform tasks that do not have any data dependencies among them. Task parallelization is then implemented on these segments of the code, wherein they are broken down into smaller chunks which are then run in parallel to fully utilize all the cores in a system, and improve the overall efficiency. Experiments are carried out to check the execution times of the algorithm when implemented with varying sizes of the location datasets and connectivity densities among the locations. Results are plotted for execution times and the speedup in each case. Cab\_Share is implemented in Java and the Java Concurrency API is used to introduce parallelization.

# Introduction

Cab\_Share is divided into three modules – the **Controller**, the **Modeller**, and the **Input/Output**.

## Controller

The controller module contains classes that take care of the working of the algorithm. The Main class is where the computation starts. It calls functions to take inputs from an input text file, process this input, and to load the data from the input into the respective data structures via the InputHandleController. It then calls functions to find the shortest path and perform other processing and scheduling operations included in the algorithm via the ProcessController. It also takes care of calling the function that writes the processed output to the output file.

The pseudo code for the overall algorithm is as follows,

|  |
| --- |
| // In Main   * Read the input file and load the data into the respective data structures // via InputHandleController * Find the shortest path between any two set of nodes in the adjacency matrix // via ProcessController  1. implement Floyd-Warshall and update matrix with shortest paths 2. also update the path matrix which will provide the resulting path between any two nodes  * Calculate the revenue for each request // ProcessController * Sort the requests according to their revenues and mark all the high revenue requests // ProcessController * Start clock with currentTime = 1, and   While (currentTime < endOfDay), do     1. Get all the valid requests at the time //ProcessController 2. Check suitable cabs for the valid requests //ProcessController 3. For each request check the available cabs 4. Select the best cab to service the request 5. If there’s no suitable cab, leave the request 6. Update the details for each cab // ProcessController 7. currentTime = currentTime + 1   end   * Print results to output file // ProcessController |

Inside the ProcessController, are also present functions - replaceToInfi, list2Matrix, and printResults. These are utility functions that traverse a part of, or in some cases, the complete input data matrix to carry out the computations that they are designed for. It is for this high data traversal functionality that these became potential constructs to be parallelized to improve the efficiency of the code. As a result, the functions parallelizeReplaceToInfi, parallelizeList2Matrix, and parallelizePrintCabResults were introduced inside the ProcessController itself, which are parallel implementations of replaceToInfi, list2Matrix, and a part of printResults respectively.

The Controller also contains the classes ShortestPath, ParallelShortestPath, and InputGenerator.

**ShortestPath**, as the name suggests, is used to calculate the shortest distances between nodes stored in the adjacency matrix created from the input data. This class contains the serial implementation of Floyd-Warshall, and also, as an option, a call to the parallel implementation of the algorithm via **ParallelShortestPath**.

The **InputGenerator** is a file created to allow a user to generate random input matrices which can be used to test the algorithm. The user can decide the size of the matrix and how densely it gets populated.

## Modeller

The Modeller module contains classes that take care of the storage of the segregated input data into various data models. The description of the models and the type of data they store is as follows,

RequestModel

|  |
| --- |
| * reqID // Request ID based on the revenue of the request * origin // origin of the request * dest // destination of the request * tofReq // time when request was placed * tofExp // time of expiration of request * length // length of the request (default 2) * rev // revenue of the request (default 0) * status // status of the request (open/close) ("O"/"C") * pic // the time at which requester is picked up at source * drop // the time at which requester is dropped at destination |

CabLocationModel

|  |
| --- |
| * cab // a number given to each cab * location // location of the cab at its timestamp * occupancy // number of passengers in the cab at the time * timestamp // timestamp of the cab * revenue // revenue earned by the cab * nextDest // the location where the cab is headed next * finalDest // the location where the cab has to finally reach * km // kilometers traveled by the cab * status // status of the cab * timer // the amount of time the cab is sitting idle |

CabShareModel

|  |
| --- |
| * RequestModel * CabLocationModel * occupancy * status |

## Input/Output

* the input is a text file that contains information about the number of locations, number of cabs, capacity of each cab, and the number of requests to process and their details
* the output is a text file that provides information about the requests picked and dropped by each cab, the revenue earned by each cab, total requests served, and the total revenue earned

# Parallelization and Results

As already discussed, four main constructs are identified and parallelization implemented on them. These are,

* the implementation of Floyd-Warshall to find the shortest path matrix
* the function list2Matrix
* the function replaceToInfi
* the printing of individual cab results

## Configuration

* The system: Intel Core i7 CPU @ 2.50 GHz

Cores: 4

Logical Processors: 8

* Default number of threads used = 8

Computations were also performed with other values of number of threads. Higher values than 8 did not show any improvements in the outcome because of the limitations of the system.

* To run the main function in class Main, provide the two arguments,

>> java Main <input.txt> <output.txt>

## Results

Cab\_Share was run with a varied set of inputs, wherein the number of location nodes, number of cabs, and the number of requests were changed for each set. The time elapsed (in milliseconds) was recorded for each construct under each input set and also for the overall execution. The results that were gathered are as follows,

**Input 1**

Number of location nodes: 5

Number of cabs: 10

Capacity of cabs: 5

Number of requests to process: 20

**Time Elapsed** (milliseconds)

|  |  |
| --- | --- |
| Serial Implementation | Parallel Implementation |
| |  |  | | --- | --- | | list2Matrix | 0 | | |  |  | | --- | --- | | list2Matrix | 4 | |
| |  |  | | --- | --- | | replacetoInfi | 0 | | |  |  | | --- | --- | | replacetoInfi | 1 | |
| |  |  | | --- | --- | | ShortestPath | 0 | | |  |  | | --- | --- | | ShortestPath | 1 | |
| |  |  | | --- | --- | | printResults | 3 | | |  |  | | --- | --- | | printResults | 7 | |
| |  |  | | --- | --- | | Total execution time | **166** | | |  |  | | --- | --- | | Total execution time | **199** | |

**Input 2**

Number of location nodes: 100

Number of cabs: 10

Capacity of cabs: 5

Number of requests to process: 1000

**Time Elapsed** (milliseconds)

|  |  |
| --- | --- |
| Serial Implementation | Parallel Implementation |
| |  |  | | --- | --- | | list2Matrix | 1 | | |  |  | | --- | --- | | list2Matrix | 5 | |
| |  |  | | --- | --- | | replacetoInfi | 0 | | |  |  | | --- | --- | | replacetoInfi | 2 | |
| |  |  | | --- | --- | | ShortestPath | 8 | | |  |  | | --- | --- | | ShortestPath | 48 | |
| |  |  | | --- | --- | | printResults | 36 | | |  |  | | --- | --- | | printResults | 35 | |
| |  |  | | --- | --- | | Total execution time | **687** | | |  |  | | --- | --- | | Total execution time | **705** | |

### Observation 1

With input matrices of sizes up to 100, the overall performance is degraded when parallelization is implemented. The execution times go up in each case. The reason being that for smaller inputs, the overhead introduced by implementing parallelization surpasses the reduction in execution time that parallel processing offers. As a result, the effective running time is increased. Parallelization has worse effect when the input matrix is of size 5 as compared to when it is of size 100.

**Input 3**

Number of location nodes: 500

Number of cabs: 10

Capacity of cabs: 5

Number of requests to process: 1000

**Time Elapsed** (milliseconds)

|  |  |
| --- | --- |
| Serial Implementation | Parallel Implementation |
| |  |  | | --- | --- | | list2Matrix | 9 | | |  |  | | --- | --- | | list2Matrix | 8 | |
| |  |  | | --- | --- | | replacetoInfi | 4 | | |  |  | | --- | --- | | replacetoInfi | 4 | |
| |  |  | | --- | --- | | ShortestPath | 229 | | |  |  | | --- | --- | | ShortestPath | 218 | |
| |  |  | | --- | --- | | printResults | 37 | | |  |  | | --- | --- | | printResults | 30 | |
| |  |  | | --- | --- | | Total execution time | **1749** | | |  |  | | --- | --- | | Total execution time | **1520** | |

**Input 4**

Number of location nodes: 1000

Number of cabs: 10

Capacity of cabs: 5

Number of requests to process: 1000

**Time Elapsed** (milliseconds)

|  |  |
| --- | --- |
| Serial Implementation | Parallel Implementation |
| |  |  | | --- | --- | | list2Matrix | 19 | | |  |  | | --- | --- | | list2Matrix | 14 | |
| |  |  | | --- | --- | | replacetoInfi | 8 | | |  |  | | --- | --- | | replacetoInfi | 7 | |
| |  |  | | --- | --- | | ShortestPath | 1675 | | |  |  | | --- | --- | | ShortestPath | 1259 | |
| |  |  | | --- | --- | | printResults | 28 | | |  |  | | --- | --- | | printResults | 19 | |
| |  |  | | --- | --- | | Total execution time | **3378** | | |  |  | | --- | --- | | Total execution time | **2845** | |

### 

### Observation 2

For input matrices of sizes 500 and 1000, parallelization shows positive results and the overall execution time is reduced for each identified construct. The calculation of the shortest path shows the most significant results. For each outer ‘k’ loop in Floyd-Warshall, the inner ‘i’ and ‘j’ loop computations are divided into segments; the number of segments being equal to the number of threads used (in our case, 8). Higher number of threads (for example, 16) were also used, but on our 4 core system, the performance doesn’t show any improvement.