Parallelization of Cab-sharing and Scheduling problem

Project Report

for

CMPT 885

(Summer 2016)

* Eshan Raina

301260346

**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 corresponding speedups 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 load the data from the input into 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 to write 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,