**Project 3: Google Flights Search Engine**

**Setup**

The proj3 directory has the following directory structure (bolded indicates directory):

Note: The Go files weren’t put into a separate src directory because the import package dependencies didn’t run correctly when structured that way.

* **proj3**
  + **balancing (work balancing)**
    - balancing.go
  + **bookings (sequential and parallel functions)**
    - bookings.go
  + **deque** 
    - deque.go
  + **flights (main function taking input files)**
    - benchmark.py
    - flights.go
    - maps25000.txt (and other datasets)
    - queries25000.txt (and other datasets)
  + **futures**
    - futures.go
  + **generate (used to generate input)**
    - cities.json
    - generate.py
  + **go.mod**
  + **graph** 
    - graph.go
  + **heapq (used to implement Dijkstra in graph package)**
    - heapq.go
  + **queue**
    - queue.go
  + Project 3.pdf
  + README

To run the program in **Python** to get the speedup graph, go into the **flights** folder, then run:

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**$ ./** **python benchmark.py**

To run the program in **Go**, go into the **flights** folder, then run the command:

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**$ ./** **go run flights.go 6 map.txt queries.txt**

Usage Statement

Usage: flights <number of threads> <maps.txt> <queries.txt>

<number of threads> = the number of threads to be part of the parallel version

no input indicates sequential version

<maps.txt> = file with available flights and their prices

<queries.txt> = file with customer requests indicating source and destination

**Task**

This system implements a flight search engine where it returns the cheapest flights with the given source and destination location for every customer request.

The program reads in two files:

1. maps.txt – a JSON file with all of the available flights and their prices used to make a weighted directed graph
2. queries.txt – a JSON file with the customer requests indicating their source and destination locations

Maps has the following format:

{

“origin”: “Chicago,

“destination”: “New York”,

“price”: 250

}

Queries has the following format:

{

“id”: 12534,

“origin”: “Chicago”,

“destination”: “London”

}

For each request, the server will output a response that includes the corresponding id number, origin and destination locations, total price, and shortest path.

{

“id”: 12534,

“origin”: “Chicago”,

“destination”: “London”,

“price”: 850,

“path”: [“Chicago”, “New York”, “London”],

}

**Part 1: Graph generation**

The program first creates a directed weighted graph given the inputs from Maps. The JSON objects from maps.txt are decoded and created into a slice of objects called **Flight** where **Flight** takes the form of:

**type** Flight **struct**{  
 Origin string  
 Destination string  
 Price int  
}

The program then takes the slice of **Flights** and adds an edge into the graph for each **Flight** it processes. After adding all the edges, the program executes Dijkstra’s algorithm for each city in the graph.

**Part 2: Parallel All Pairs Shortest Path Algorithm[[1]](#footnote-1)**

The all pairs shortest path algorithm finds the shortest path between all pairs of nodes in the graph. Dijkstra’s algorithm returns an array of the shortest weighted path from the origin to every node. The implementation executes Dijkstra’s algorithm n times for n number of cities (nodes).

The information for the shortest path is stored in an object called **MinPath** that takes the form:

**type** MinPath **struct**{  
 Source string  
 Destination string  
 Price int  
 Path []string  
}

The **MinPaths** are stored in a matrix of indices i, j where i represents the index of the source city and j represents the index of the destination city.

In the given example below, x (0, 2) holds the **MinPath** for Chicago to London.

Matrix = [[][][x],

[][][],

[][][]]

Index = {

“Chicago”: 0,

“New York”: 1,

“London”: 2

}

In the sequential version, this is executed via iterating through every node. In the parallel version, this is executed via threads and **futures**. The matrix is a nested array of futures. After completing graph creation, the nodes (cities) are enqueued into a shared queue where each thread is responsible for a set of cities as the source node. Each thread dequeues an equal fraction of the queue into their own private queue and begins Dijkstra’s process on the popped node from their private queue. The thread then **sets** the corresponding section of the matrix with an array of futures. The program **gets** the future in the query processing section later when it retrieves the future from the matrix.

**Part 3: Query Processing**

After creating the matrix of cheapest paths, the program processes the inputs from Queries file and returns a queue of **Request** objects which takes the form of

**type** Request **struct**{  
 ID int  
 Origin string  
 Destination string  
}

The stream of **Requests** is then processed so take the information from the **MinPath** of the corresponding source and destination locations from the matrix and outputs a **Result** object in the form of

**type** Result **struct**{  
 ID int  
 Origin string  
 Destination string  
 Price int  
 Path []string  
}

The **Result** is then encoded as output.

In the sequential version, the program iterates through the series of inputs from Queries, decodes them into Requests, processes the Request into a Result, then encodes the Result. In the parallel version, the program iterates through the Queries inputs and enqueues the Requests into a shared queue. A set of threads then dequeues an equal fraction of the queue into their own private deque and begins processing the Requests. A wait group is put in place to ensure all the threads finish before terminating the program. After every Request processed, the thread checks for **rebalancing**. If the work difference between that thread and another thread is greater than 5 (arbitrarily chosen threshold), then the thread steals Requests from the dequeue until the deque length between the two threads are equal.

**Hotspot and Bottlenecks**

The three hotspots are generating the graph, running the all pairs shortest path algorithm to create the matrix, and processing the requests. Graph generation cannot be parallelized due to complexities of concurrently adding nodes and edges to the same graph. The data dependencies and having to lock and unlock certain nodes while checking if nodes or edges existed would have led to multiple race conditions. Therefore, parallelizing this part was not attempted. Because this part was not parallelized and is also a hotspot, graph generation is a large bottleneck in the program.

The graph also got proportionally bigger as the number of queries increased. In order to fulfill the growing number of *unique* queries, the size of the graph must also increase to fulfill those requests. Therefore, the number of unique edges needed to be added to the graph were proportional to the number of query requests.

|  |  |
| --- | --- |
| **Number of Queries** | **Number of Flights (Edges)** |
| 25000 | 25959 |
| 50000 | 51337 |
| 75000 | 76650 |
| 100000 | 101909 |

The all pairs shortest path algorithm was parallelized by concurrently running Dijkstra’s algorithm on all of the nodes and simultaneously writing to the shared matrix. There is no data dependency between running Dijkstra’s on one node verses another node since threads are only reading the graph and not writing to it. There is also no overlap when writing to the shared matrix since each thread is responsible for a non-overlapping set of source nodes, so the row for each source node is only written to once. Having no data dependency allows this part of the program to be parallelized.

The request processing was also parallelized and enhanced using a **work balancing** algorithm. Since the queries were completely data independent, the data decomposition achieved by splitting up the queries led to speedup. In addition, work balancing allowed for shorter idle periods while in the wait group. Work balancing was checked after each thread popped one request from their private deque to see if there was a significant work discrepancy between them and another thread.

**Challenges**

Since futures and work balancing was new, it was challenging to conceptually understand how they worked and interacted with the broader design of the program. A lot of time was also spent on designing, then redesigning the details of the program. The project required implementing a large range of data structures, which was unexpected but rewarding. The most difficult part of the assignment was similar to that of Project 1 which was debugging race conditions. The unexpected race conditions came from multiple threads enqueuing and dequeuing deque and queue. The work balancing algorithm required implementing Lock and Unlock methods on deques that allowed one thread to access the deque at once. This implementation was later also added to queue structure so that only one thread can dequeue from the shared queue at once while splitting up work. The Lock and Unlock methods were executed surrounding any modification to the data structures.

**Specifications of Testing Machine**

OS: MacOS Big Sur

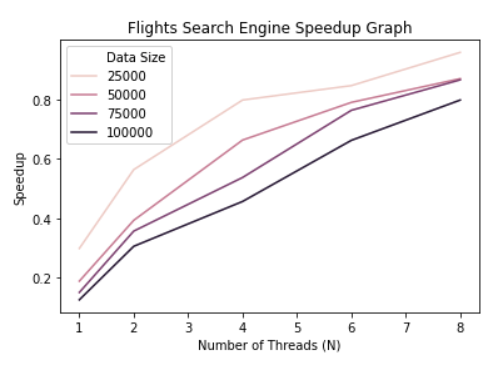
Version: 11.0.1

Machine: MacBook Air (13-inch, Early 2015)

Processor: 1.6 GHz Dual-Core Intel Core i5

Memory: 8 GB 1600 MHz DDR3

**Speedup**

****

The time recorded for each data size and thread combination took the average of 5 runs. From the graph, the speedup is pretty linear to the number of threads. There’s no noticeable speedup plateau, although there’s a mild one at threads = 4 for the 25000 and 50000 data sizes. The program creates the graph sequentially first, then parallelizes graph traversal and query searches. Since the graph gets proportionally bigger to the number of queries, graph traversal will also take longer. If all three steps were parallelized as opposed to the latter two, there might be a more noticeable plateau since the sequential part might have buffered out the overall speedup.

**Appendix**

**Input Generation**

The input for each file size is two files, map.txt and queries.txt, in JSON format. The python and JSON file generating this input is in the generate folder.

* **generate**
  + generate.py
  + cities.json
* generate.py takes cities.json as an input to create a map and query file based on the number of requests.
* cities.json is a JSON file of a list of all the cities in the world. It was taken from

<https://github.com/lutangar/cities.json/blob/master/cities.json>

To run the program in **Python**, go into the **generate** folder, then run:

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**$ ./** **python generate.py cities.json maps250.txt queries250.txt 250**

Usage Statement

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Usage: generate.py cities.json <maps.txt> <queries.txt> <number\_queries>

cities.json = json file of city list

<maps.txt> = file name for empty txt file to put flights info

<queries.txt> = file name for empty txt file for customer requests

<number\_queries> = total number of customer requests to put into queries.txt file

Sample Runs:

./generate cities.json maps250.txt queries250.txt 250 -- Generates 250 queries

1. https://en.wikipedia.org/wiki/Parallel\_all-pairs\_shortest\_path\_algorithm [↑](#footnote-ref-1)