**Multicore Programming**

**Project Clojure**

**Bilal Laaroussi El Alami**

**0547571**

*2. Implementation: Describe the implementation on an abstract level. Use illustrations or code snippets to guide the reader where necessary. Answer the following questions: Which concurrency mechanism(s) (e.g. refs, agents, atoms) did you use and why? How did you parallelize the application (i.e. which mechanism did you use to create multiple execution threads)? How do you ensure correct, thread-safe concurrent access to the system? What are the potential performance bottlenecks?*

Data representation

Flights are atoms, the content of the atom is called the flight-data which has the form

{:id id :from from :to to :carrier carrier :pricing pricing})

We used atoms to represent a single flight because it prevents data races, which makes it easier to avoid getting to invalid states like overbookings etc.

The *initialize-flights* function is going to return two maps. The first one groups flights that have the same *from* and *to* value, the second map groups flights of the same carrier. An example.

A graph paper with writing on it

Description automatically generated

Representing the data this way allows to easily get for example all flights from BRU to ATL or to get all flights from the carrier DELTA. This is important as how we essentially book a flight is by looping over all candidate flights and try to book each one, once we get a positive result we have successfully booked a flight, therefore it is important to filter the candidate flights as much as possible to have a good efficiency because then we would loop over less flights. Grouping the flights by carrier allows to easily get all flights of some carrier, which makes it easier to do a sale.

The update-flight function is important for understanding the application:

(def reasonBooking "reasonBooking")

(def reasonSale "reasonSale")

(def carriers-undergoing-sale (atom '()))

(defn update-flight [flight update-flight-data reason]

(loop [oldFlightData @flight]

(let [newFlightData (update-flight-data oldFlightData)]

(if (and (= reason reasonBooking)(some (fn [carr] (= carr (@flight :carrier))) @carriers-undergoing-sale))

;this means the flights of the carrier are undergoing a sale, to get atomic/consistent results we recur

(recur @flight)

;if the newdata is not valid (overbooking, negative price, ...) we dont update. extra safety mechanism because update-flight-data function should not return corrupted data in the first place

(if (not (validate-flight newFlightData))

[oldFlightData, oldFlightData]

(if (compare-and-set! flight oldFlightData newFlightData)

[oldFlightData, newFlightData] ;returning the old an new data so the user can confirm if a change really happened

(recur @flight)))))))

The *compare-and-set!* Ensures that the data hasn’t changed in the instant of updating it. This is crucial as it is that that prevents data races and bad interleavings, and thus prevents reaching a corrupt states like overbookings, etc.

The carriers-undergoing-sale variable stores the carriers that are currently undergoing a sale i.e., the carriers whose flights are **in the process** of being discounted. So if for example “DELTA” is in that list it means that it is possible that there are some flights of delta that are discounted, and others not.

When trying to book a flight but the carrier of that flight is undergoing a sale (it is in the carriers-undergoing-sale list) the update-flight will recur, this until the sale is over. This ensures that when a carrier does a sale it will not be possible for some customers to book a discounted flights and at the same time other unlucky customers to book a yet undiscounted flight.

This influences performance because there will be more loops of update-flight but the process of discounting flights is an operation that does not take a lot of time, and also only happens occasionally so this is not too bad.

The *book* function takes as input one flight atom and one customer, it is going to attempt to book that flight for that customer and returns whether that customer was successful booking that flight, and the price the customer paid if the booking happened. The implementation is simply a call to update-flight with a reason of *reasonBooking*, a customer, and a function that updates the flight-data customized to the customer.

A customer is a map {:id id :from from :to to :seats seats :budget budget}

The *find-and-book-flight* function takes as input a list of flights grouped by *from* and *to*, and one customer, it is first going to get a list of candidate flights that match witch the *from* and *to* values of the customer, so these flights already pass this criteria. Then it is going to loop over those flights and try to book a flight one by one. Once a positive result is found, we stop, and the customer was successful booking a flight.

We loop in a sequential way over the collection of candidate flights, but it would be more efficient if every call to *find-and-book-flight* traverses the collection in each time a different random way, this way there would be less customers trying to book the same flight at the same time so the *update-flight* function would recur/retry less.

The smaller the collection of candidate flights is, the lower the latency of the requests, this because less tries of bookings happens.

We could make the collection of candidate flights even smaller by grouping the flights by *from*, *to* **and** a minimum price (which would be the discounted price of the cheapest travel class). So when a customer tries to book a flight we can in advance filter out flights that are too expensive in whatever circumstance.

The *process-customers* function is going to call *find-and-book-flight* in parallel on all the customers. Parallelism is achieved by the *coarse-pmap-threads* function explained later.

Starting or ending a sale is simple. We add the carrier to the list of carriers that are undergoing a sale. Then for all flights a call in parallel (through the *coarse-pmap-threads* function) to update-flight is done with as argument a reason of *ReasonSale* and an update function that discounts the travel-classes of the pricing of the flight data.

The function *coarse-pmap-threads* is the function that gets us parallelism.

(defn coarse-pmap-threads [fun args]

(let [size (quot (count args) @number-threads) ;;size of list that will be in one future

parts (split args size)]

(doall (pmap (fn [part] (doall (map fun part))) parts))))

The *split* function splits a list in N equal parts. The rest of the function is self-explanatory. pmap is implemented with futures so each part will be processed in a separate thread.

This function is used by the *sale*, and *process-customers* functions.

**Evaluation**

We evaluated correctness through different tests.

* The *flight-test* ensures that illegal updates of flight data of any nature are impossible.
* The *book-test* verifies that a customer can book a flight that should be bookable for him, and that a customer can’t book a flight that should not be bookable for him.
* The *find-and-book-flight-test* verifies that a customer can find and book a flight.
* The *test-booking* test verifies that a flight can be booked by multiple customers.
* The *test-no-overbooking* and *test-no-overbooking-2* tests verify that no overbookings are possible, this by creating a higher demand of customers than a supply of flights.
* The *test-sale-consistency* test verifies that it's not possible for some customers to see a discounted flight and other customers to see an undiscounted flight. We do this by processing a set customers that all have the same amount of money. At the same time the sales process starts, after the customers are processed we verify that all customers have paid the same price (check the function for more details)

Experiments

Experimental setup

* **Apple M1 chip**  
  8-core CPU with 4 perform­ance cores and 4 efficiency cores  
  7-core GPU, 8-core GPU  
  16-core Neural Engine
* Memory: 8GB unified memory
* OS: macOS Monterey 12.2.1

Clojure version 1.11.1

openjdk version 20.0.2

Charger connected, closed all other programs/applications/processes.

The *input-experiments.clj* file defines 200 airport-codes and 20 carriers. The function *flights(n)* generates n random flights from a random departure airport to a random destination airport through a random carrier. The *customers(n)* function generates n random customers with a random *from* airport to a random *to* airport, seats between 1 and 4 and a budget between 95 and 800.

Each experiment is ran 30 times

The *run-process-once* function processes customers and starts a sale process in new threads. It times the time it took for processing of the customers. We define the configuration by providing the flights, customers, time of sales, time between sales as arguments.

The *experiment-speedup-threads* experiment measures the time to process all customers, the independent variable is the number of threads.

The input arguments for run-process-once, the configuration:

10 000 random generated flights

100 000 random generated customers

2000 as time between sales

50 as duration of a sales

Results

Conclusion