VRIJE
UNIVERSITEIT
BRUSSEL

CLOJURE PROJECT

A Multicore Programming Project

Patrick Wendo

June 2, 2024

Sciences and Bio-Engineering Sciences

1 Introduction

The task at hand was to implement a parallelization of an e-commerce system for which a sequential implementation had been provided. We were to use the concurrency mechanisms taught in class in order to ensure correctness of data manipulation. In my implementation, the first step was to identify where conflicts would arise and I would end up with corrupt data. I came up with the following table to guide me.

Store Access	Sale	No Conflict	Conflict
Different	No	✓	-
Different	Yes	-	✓
Same	No	-	✓
Same	Yes	-	✓

Table 1: Conflict Map

This table says hat when the stores being accessed are different and there is no sale, then there is no conflict. However, if the stores are the same, or there is a sale happening concurrently with a process customer transaction, then there may be a conflict. The goal is to handle these conflicting cases in particular. In my implementation I used refs as I find that their synchronized and coordinated nature of managing state rather important in this exercise.

With regards to experimentation, I focused on the number of threads being used by the parallel implementation, the number of customers, and sale duration. I varied the number of threads between 2, 4 and 8, the number of customers from 50 to 2000, and the sales periods between 2 to 50 for time between sales and 5 to 20 for time of sales.

2 Implementation

In the implementation, I used refs as mentioned earlier as I wanted to leverage the synchronization and coordination in order to ensure correctness of the implementation. Particularly, the prices and the stock are what I changed to be refs. This is because during a sale, the prices are what change and this could be in conflict with an ongoing customer process. As such we want to ensure that if the price changes, then the customer transaction would abort and restart. Similarly for stock, I made it a ref because multiple customers may access this at the same time and I wanted to ensure that one would abort and retry later.

The entire `process-customer` block is wrapped in a dosync to ensure it fully aborts without committing its changes if it has to. The `process-customers` function on the other hand will always submit a new task to the thread pool. This task will be the `process-customer` task for each new customer. I use a fixed thread pool defined in the main function as shown below. This allows me to fix and vary the number of threads for bench-marking in order to find the optimal number of threads.

```
(defn main [threads]
    (def pool (Executors/newFixedThreadPool threads))
    ...
```

In order to ensure correctness, I posed a couple of tests to my implementation. The first was to ensure that if there are not enough items in stock, the transaction does not alter the stock. For instance, if our stock looks like this

```
(def stock
        ; Aldi
    [
        [14] ; Apple
        [2] ;Avocado
   ]
)
```

And our customers look like this:

```
(def customers
    [
        {:id 0 :products [["Apple" 7] ["Avocado" 3]]}
        {:id 1 :products [["Apple" 9] ["Avocado" 3]]}
    ]
)
```

It is clear to see that neither customer can be processed because they both need 3 avocados when there are only 2 in stock. In my implementation, the final stock is always the same as the initial stock because none of the customers attained what they were looking for.

The second test I performed to ensure correctness was that if there are 2 customers and one product and store such that only one customer can be serviced, then only one customer should be serviced. This ensures that we never end up with negative stock. Consider if we have stock that looks like this

```
   (def stock
        ; Aldi
    [
        [14] ; Apple
   ]
)
```

and our customers look like this

```
(def customers
    [
        {:id 0 :products [["Apple" 7]]}
        {:id 1 :products [["Apple" 9]]}
    ]
)
```

Then in this situation, if it were sequential, only the first customer would ever get processed. However in parallel, we want to ensure that only one of them will get processed. Since stock is a ref, if both customers are executed in parallel, then one of the transactions will have to abort. The final stock will either have 7 or 5 apples left in stock. This is exactly what I got in my implementation.

Finally, because the sales alter the price and we already defined the price as a ref, then the set price method also uses a dosync block to ensure that changes to price are all coordinated and no dirty reads occur.

```
(defn- set-price [store-id product-id new-price]
    "Set the price of the given product in the given store to ‘new-price‘."
    (dosync
        (alter prices assoc-in [product-id store-id] new-price)
    )
)
```

3 Performance Results

All the experiments were performed on my laptop with the specifications shown on Table 2. All the experiments were repeated 30 times.

Hardware	
CPU	Intel i5-8350U (8) @ 3.600GHz)
RAM	16 GB
Software	
OS	Manjaro Linux 6.1.85-1-MANJARO
Clojure	Clojure Version: 1.11.1.1413
JVM	openjdk 22 2024-03-19

Table 2: Laptop system specifications

3.1 Experiment 1: Simple Input Parallel vs Sequential

The first experiment was to compare the simple input for both sequential and parallel implementations. The parallel implementation was varied between 2, 4 and 8 threads as well. Other parameters are shown on Table 3

Parameter	Value
Number of Customers	10
Number of Stores	4
Number of Products	4

Table 3: Simple Sequential vs Parallel parameters

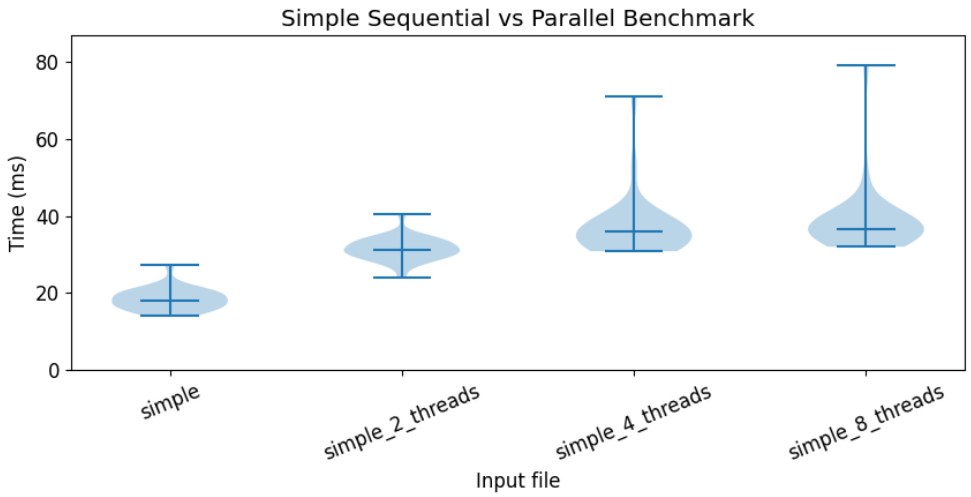

Figure 1: Sequential vs Parallel simple

The first violin plot is the sequential implementation, while the rest are the parallel implementations with variation on thread count. It would appear that the sequential implementation is faster with fewer customers to service. I would attribute this to the time taken to synchronize and coordinate the threads.

3.2 Random Input: Sequential vs Parallel, 50 customers

Parameter	Value
Number of Customers	50
Number of Stores	30
Number of Products	50

Table 4: Random Sequential vs Parallel parameters with 50 customers

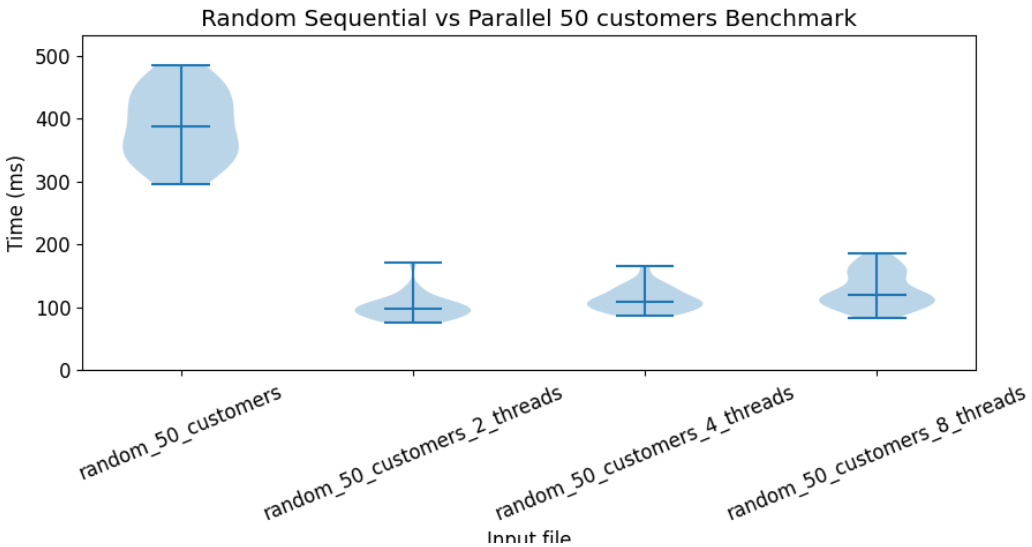

Figure 2: Sequential vs Parallel Random with 50 customers

When we scale the number of customers to 50 and increase the number of stores and products as well, we notice that the sequential implementation is markedly slower than the parallel implementation. We could in fact calculate the speed up. The mean of the sequential execution is 387.26894 while the mean of the parallel execution with 4 threads is 108.999432. The **speedup** ends up being around 3.6.

3.3 Experiment 2: Random Input: Sequential vs Parallel, 500 customers

Next we want to see what happens when we increase the number of customers tenfold.

Parameter	Value
Number of Customers	500
Number of Stores	30
Number of Products	50

Table 5: Random Sequential vs Parallel parameters with 500 customers

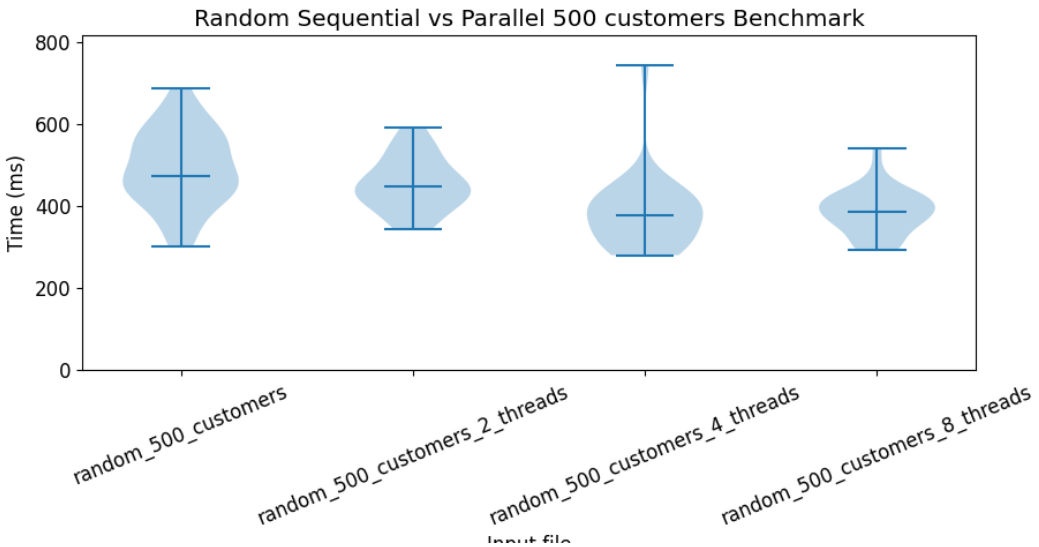

Figure 3: Sequential vs Parallel Random with 50 customers

Here we notice that our speed up slows down, to 1.2. In fact it would be best to display this as a speedup graph.

Figure 4: Speedup when we increase the number of customers

We note that the benefits of the speed up decrease with the increased number of customers. The argument I would have is the number of transactions that have to be restarted because the number of stores is not varied.

3.4 Experiment 3: Random input, varying products available

Parameter	Value
Number of Customers	500
Number of Stores	30

Table 6: Random Sequential vs Parallel parameters with varied number of products

In this experiment I am varying the number of products available at each store between 50 and 500. My reasoning is because the stock and prices are refs, then if we increase the number of products there will be more contention for the access to these. This means that it would become a good test to see how well the code handles this situation.

We find that our violin plots looks like this

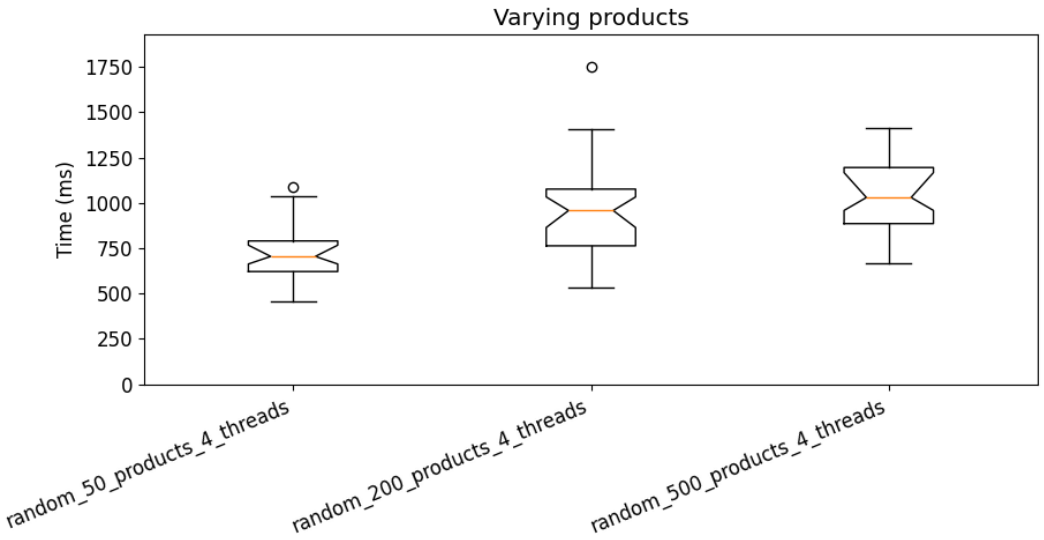

Figure 5: parallel execution time with varying product amounts

Initially my expectation was that if the number of customers was consistent, then barring the sales period, the ranges of execution time would also be consistent, regardless of the number of products. However, this is not the case. There is an almost linear association between the number of products per store and the execution time.

4 Insight Questions

The sales period depending on it's frequency and length will increase the number of retries of transactions. This is because in my implementation, the prices that are directly affected by the

sales period are refs. If these change, any concurrent transaction would have to restart. This marginally reduces throughput. If we didn't have the sales period, we could instead use atoms for the prices and reduce the number of concurrency checks we have.

If I was required to use locks as opposed to refs, The implementation would require that whenever a customer is being processed, they acquire a lock on the stock and the price. This could prove problematic because once a lock is acquired all other transactions would be waiting until the lock is released. To combat this, I would have to find some granularity at which I could balance this. There will always be a chance of deadlocks whenever sales periods happen concurrently with customer processing.