Clojure Project Report

Gakuo Patrick

May 11, 2019

1 Overview

In this experiment, we use Clojure concurrency mechanisms to simulate customers shopping in a collection of stores. We aim to maintain a consistent state where customers buy each customer buys all their products from the single cheapest store. There are two sources of inconsistency. Firstly, we do not want to result in negative stock. Secondly, we do not wish to a customer buying partially discounted products; all products must either have a discount or no discount.

We have two implementations of this shopping simulation, a parallel and a sequential application. We then conduct nine different experiments to test the performance of the parallel implementation against the sequential one. In each of these experiments, we vary one of the following:

1. The number of customers,
2. The maximum allowed stock for a product,
3. The per product maximum on a shoppinglist,
4. The number of different products sold at each store,
5. The maximum allowed shopping list size,
6. The number of stores,
7. The number of threads,
8. The time between sales,
9. The time of sales.

We then do statistics and plots to explain our findings.

2 Implementation

2.1 Parallelization approach

We have two implementations of this shopping simulation, a parallel and a sequential application. Our sequential application locks each of the stock and price lists into a single atom. On the other hand, parallel implementation locks the price list into a single atom but locks each stock item into a ref. The sequential implementation uses a doseq to process one customer at a time while the parallel implementation partitions the customer list and uses a future for each partition. This increases performance as we execute each partition in parallel.

```
1  (defn process-customers [customers]
2    (let [customer-processing-futures (atom [])
3          ;partition customers and process each partion in a
               separate
4          ;future thread
5         groups  (partition-all subscustomer-size customers)]
6           (doseq [sub-customers groups]
7            (swap!  customer-processing-futures conj
8              (future
9                (doseq [customer sub-customers]
10                 (process-customer customer)))))
11       (doseq [f @customer-processing-futures] (deref f)))
12   (reset! finished-processing? true))
```

Listing 1: Parallel process customers

2.2 Ensuring correctness

For the parallel implementation, we ensure correctness by locking the customer processing from the point of checking for available products to the buying point. Since each stock item is a ref, we use Clojure's Software Transaction Memory (STM) for thread-safe concurrency. This critical section also includes the calculation of the cheapest store and it ensures consistent use of stock. We also ensure that each transaction accesses a store's products fully discounted or not discounted by locking the entire price list with an atom.

2.3 Potential bottlenecks

In our parallel implementation, each thread has to read and write from the stock memory. With increasing threads, we are bound to have more writes blocking and a resultant rise in transaction retries. Although we reduced locking granularity to bare minimum threshold for correct execution, too many retries can hurt our performance.

3 Evaluation

As we mentioned in the overview section, we conducted nine different experiments. In each experiment the default values are:

1. number of customers = 1000
2. number of products = 1000
3. number of stores =26
4. maximum items in shopping list (shopping list size)= 100
5. maximum amount allowed per item on shopping list = 10
6. number of threads = 3
7. time between sales =50
8. time of sales =10
9. maximum stock per product =1000

In each experiment, we varied one of the above metrics and left the rest as default. We then ran the setup on both sequential and parallel implementations thirty-three times each. We did away with the first three runs on each run and did our statistics with the rest thirty runs. We report latency in milliseconds as a measure of performance. Therefore, we collected the run-time in milliseconds in our experiments. For each of the nine experiments, we have three different types of plots:

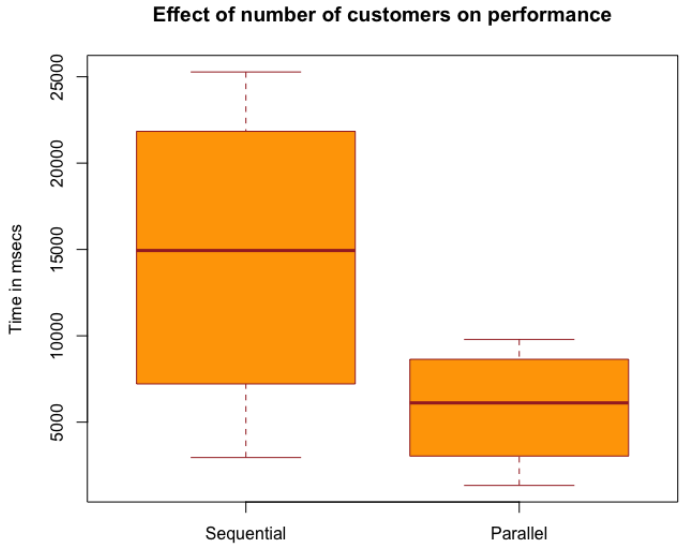
1. Box-plots to compare sequential and parallel latency
2. Line-plots to compare varying latency against each changing metric
3. Line-plots to show the change in transaction retries with the changing metrics

We ran our experiments on the below platform:

Machine	
Model	MacBook Pro
Processor	2,2 GHz, Intel corei7
Memory	16 GB 2400 MHz DDR4
OTP	version 21
Physical cpu	6
logical cpu	12

3.1 The effect of the number of customers

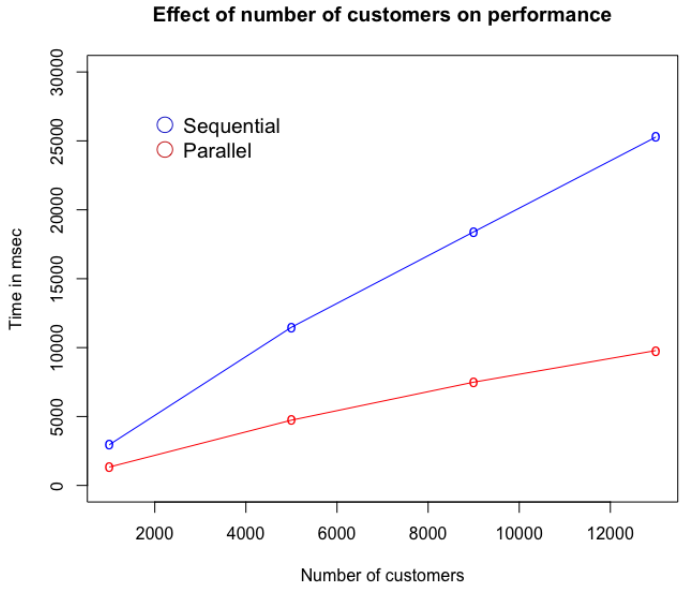
In this experiment, we kept all the defaults and changed the number of customers from 1000 to 13000 in steps of 4000. We computed the geometric means for each configuration and plotted it as follows.



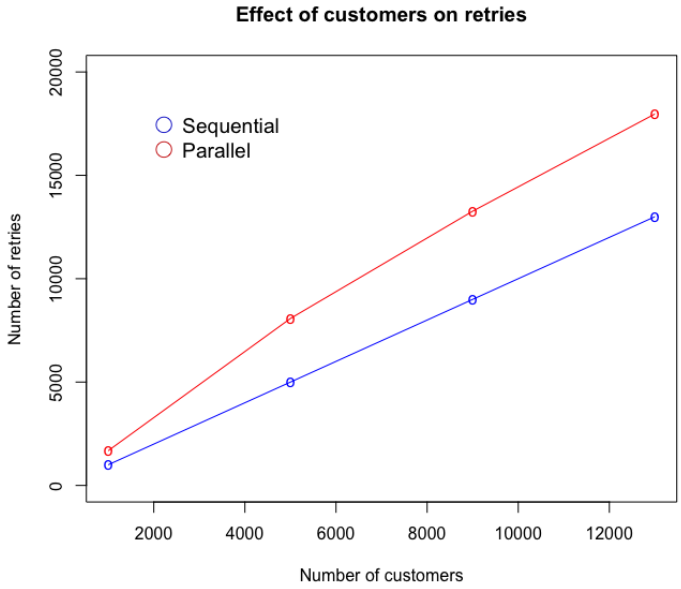
The above box plot shows that on average, the sequential implementation registered a higher latency. However, the overlap between the boxes does not allow us to rule out the possibility that the two mean latencies could be the same. Additionally, we obtained a p-value of .17 from the non-paired t-test

which does not allow us to reject the null hypothesis that states that the means are equal.

We then made a plot of run-time in milliseconds against the number of customers below. The plot shows that the parallel implementation scales better with increasing customers. That is its latency does not rise as rapidly as that of the sequential implementation.



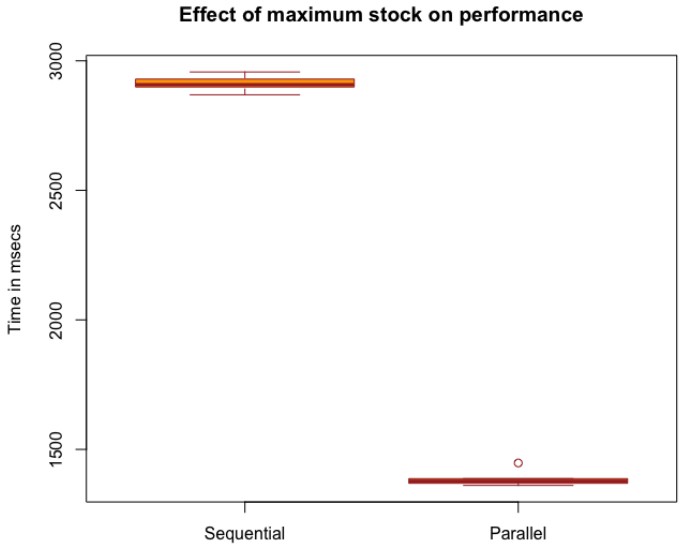
Further, we did a plot of retries against the increase in customers. As we expected, the more customers going for products in a store, the more stress on the stock. In this experiment and all the following ones, the sequential implementation registers a retry count equal to the number of customers in the system. However, we can see the difference between actual customers and transaction-retries increase with the increase in customers. This is an additional course of latency in the sequential implementation on top of the customer size.

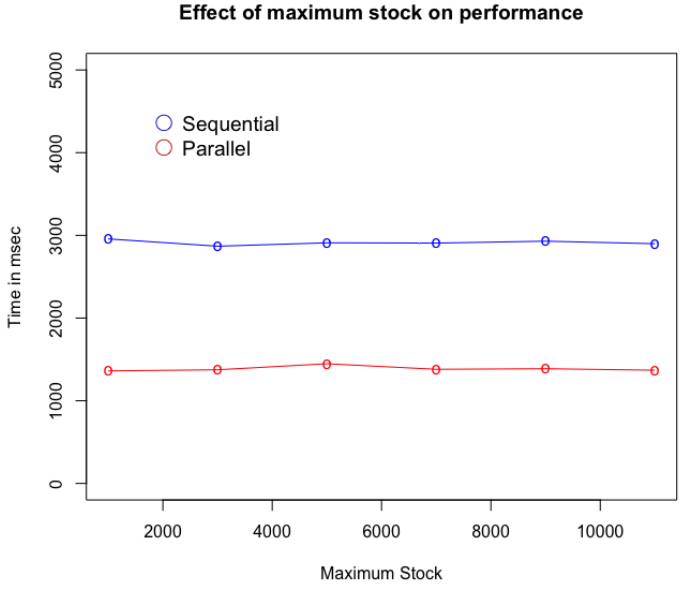


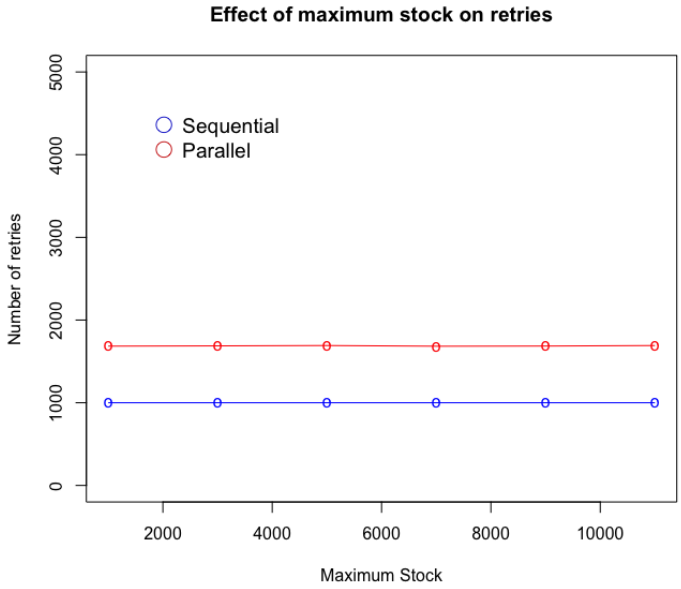
3.2 The effect of the maximum allowed stock for a product

In this experiment, we vary the maximum allowed stock for each product. We then follow similar statistical steps as in experiment one. We observe that tests run longer with more stock. We do not have customers finding inadequate stock sizes to complete their purchases. The parallel implementation keeps its latency significantly low. On the contrary, the sequential implementation scales poorly. The boxplot below shows that the two latency means are significantly different. We obtain a t-test p-value $< .5$, and we reject the null hypothesis that states the two means are the same. We also get a large Cohen-d effect-size ($< .8$).

We also show that the increase in stock results in an almost horizontal line for both retries and latency. We can intuitively tell that only those activities that directly manipulate the stock in each transaction cause a slope change in retries and latency line plots.


Effect of maximum stock on performance
Time in msecs
3000
2500
2000
1500
Sequential
Parallel

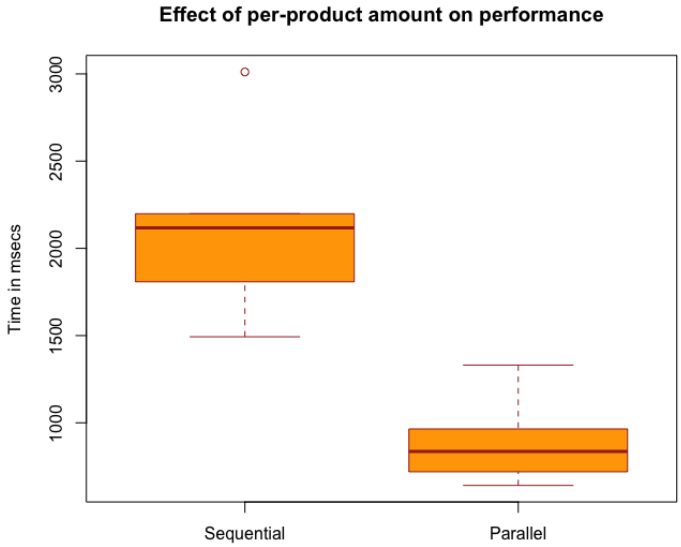

Effect of maximum stock on performance
Time in msec
5000
4000
3000
2000
1000
0
Sequential
Parallel
2000
4000
6000
8000
10000
Maximum Stock



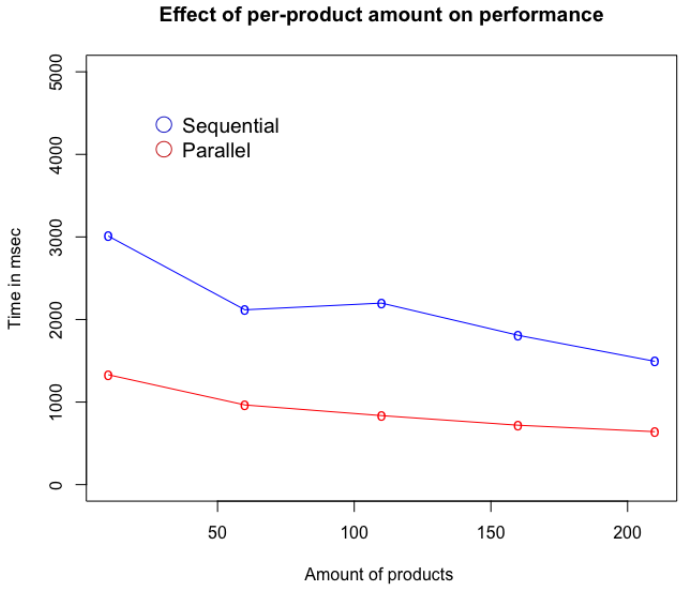
3.3 The effect per product maximum on a shoppinglist

In this experiment, we adjust the maximum amount of each product a customer can have on their shopping list. The more the customer can buy, the faster we clear stock and the following customers will not go through the buying of products steps. Therefore, we expect a decrease in latency.

Firstly, we show that the latency means between the two implementations are significantly different. We obtain p-value $< .5$ and a large Cohen-d effect-size ($< .8$). We, therefore, reject the null hypothesis.

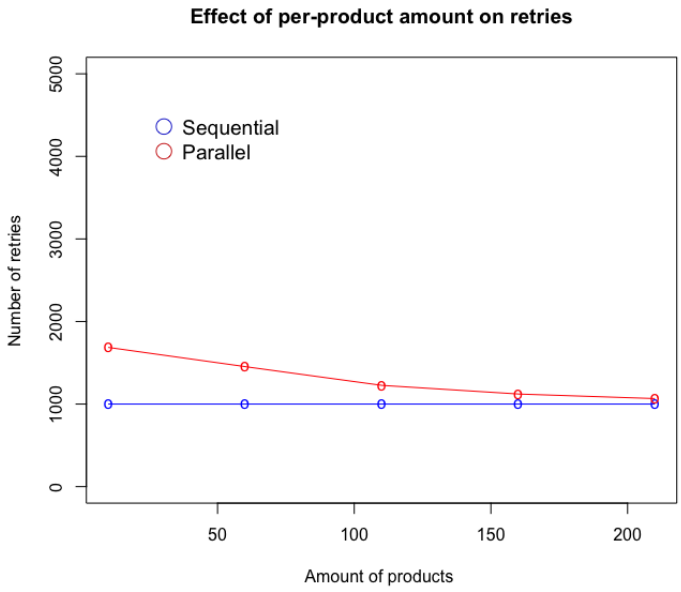


The latency plots further confirm our reasoning. With increasing possible product order sizes, latency reduces for both implementations. You now need just a few transactions to clear the stock, and the rest of the transactions will skip the purchasing steps.



Equally, we see a reduction in the number of transaction retries with increas-

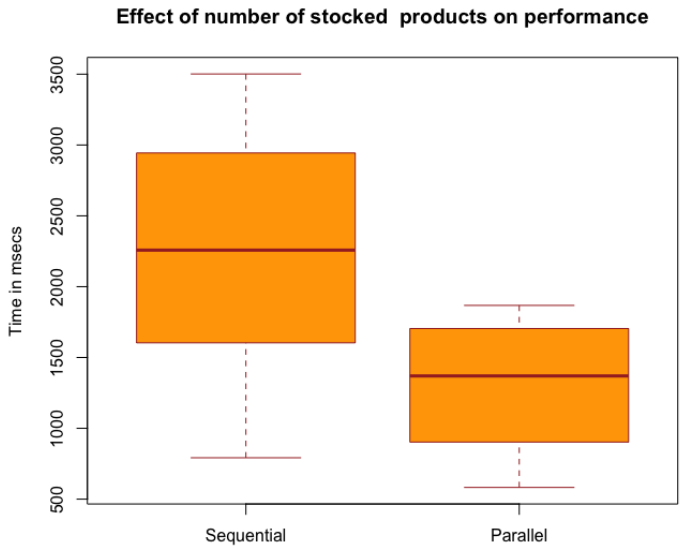
ing per-product order sizes. Fewer transactions will need to reduce stock and thus reduced retries. As expected, sequential retries remain constant.



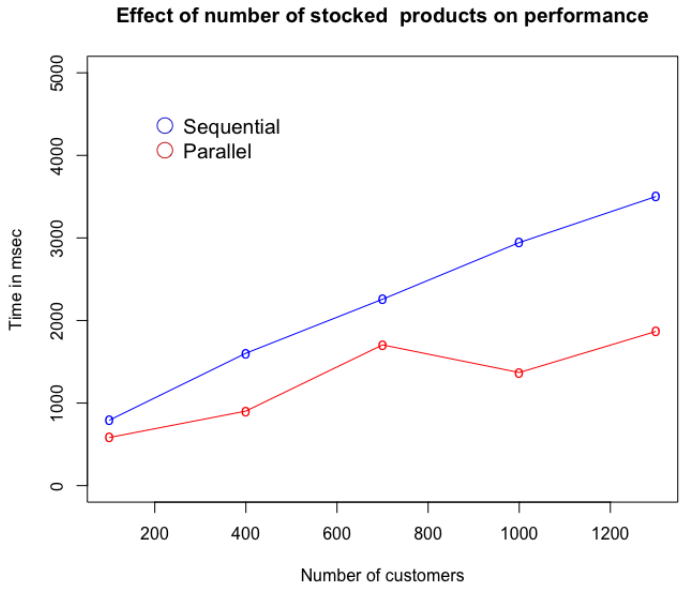
3.4 The effect of the number of different products sold at each store

In this experiment, we vary products ranges (variety). If we have more products to chose from the lower the probability that we will find stock shortages. As a result, we see a rise in latency with the increase in products choices per store.

Although there appears to be a latency difference between sequential and parallel implementations, we cannot rule out the null hypothesis. A We obtain a t-test p-value $> .5$ (.13). We also see overlap on the boxplot below.

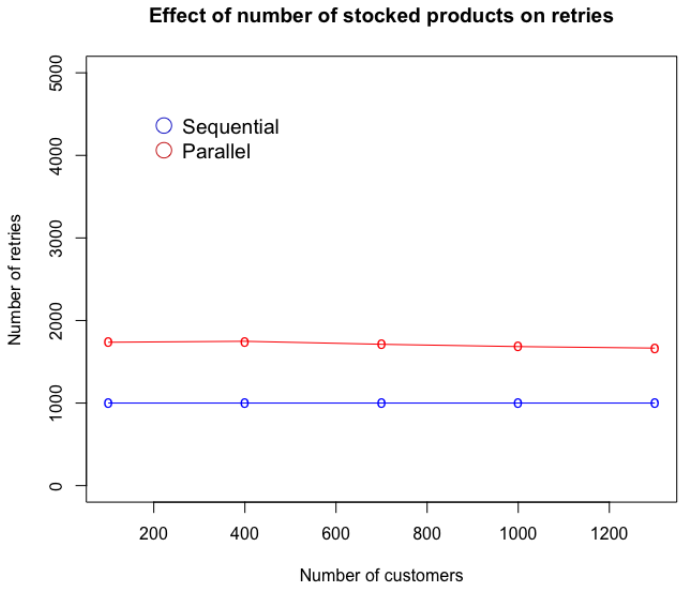


Intuitively, we see a rise in latency rises with product choices as explained above.



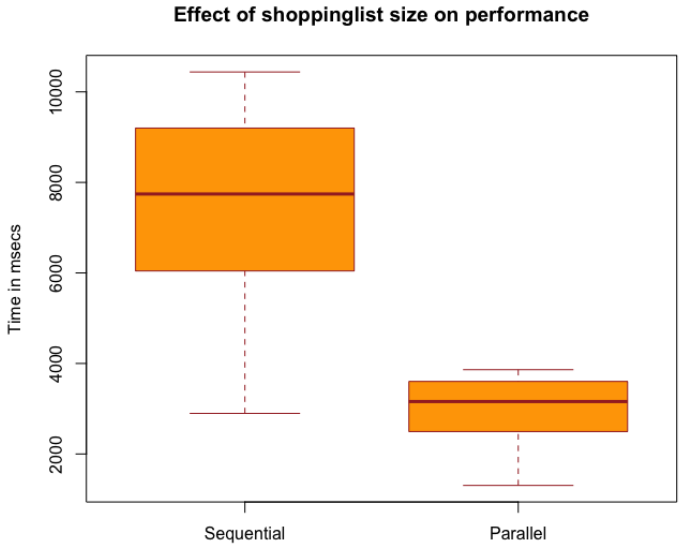
We, however, a slightly reducing retry count with an increase in the number of products offered per store. With more products, customers will have a high probability of ordering different products resulting in lower write collisions and

retries.

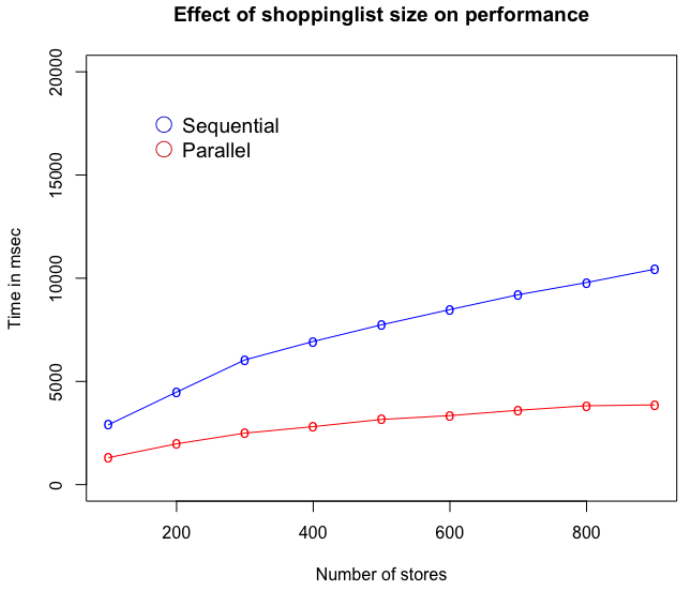


3.5 The effect of the maximum allowed shopping list size

We vary the maximum size of customer shopping lists in this experiment. With more items in a shopping list, we expect more processing and a resulting rise in latency. We obtain a t-test p-value $< .5$ and a large Cohen-d effect-size ($< .8$). We reject the null hypothesis.

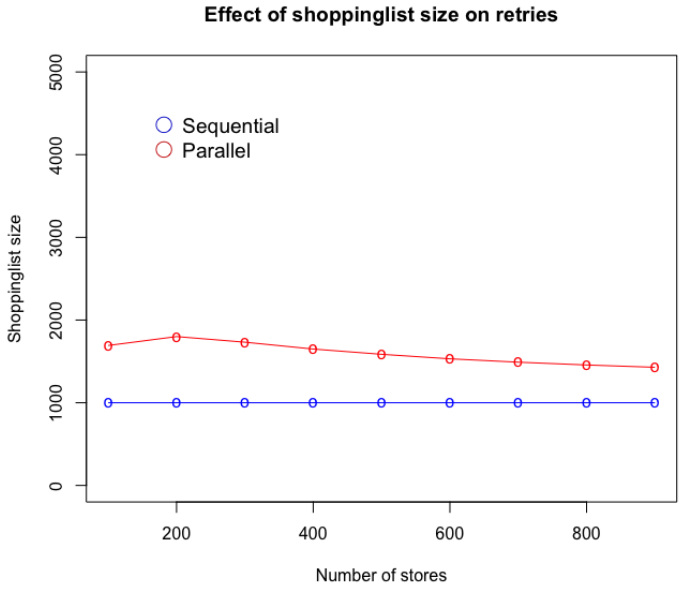


The latency intuitively rises with the sizes of shopping lists. The parallel implementation scales more gracefully with a lower slope gradient than the sequential implementation.



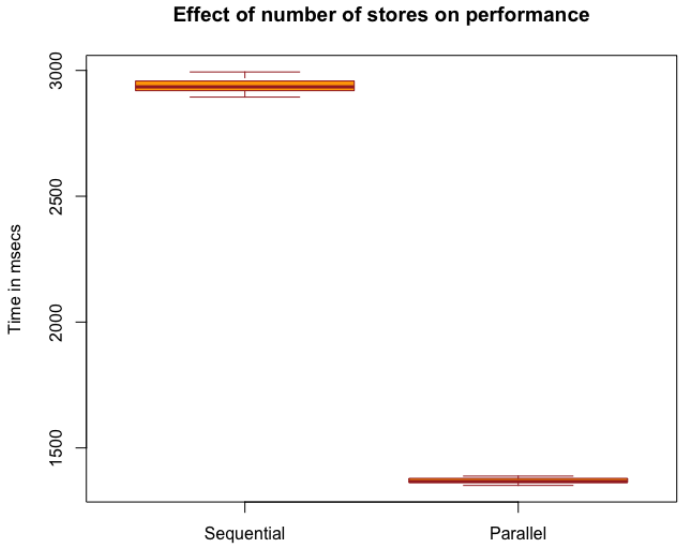
The retries initially rise and then fall for the parallel implementation. In the beginning, all products on a shopping list are available. At that point, there will

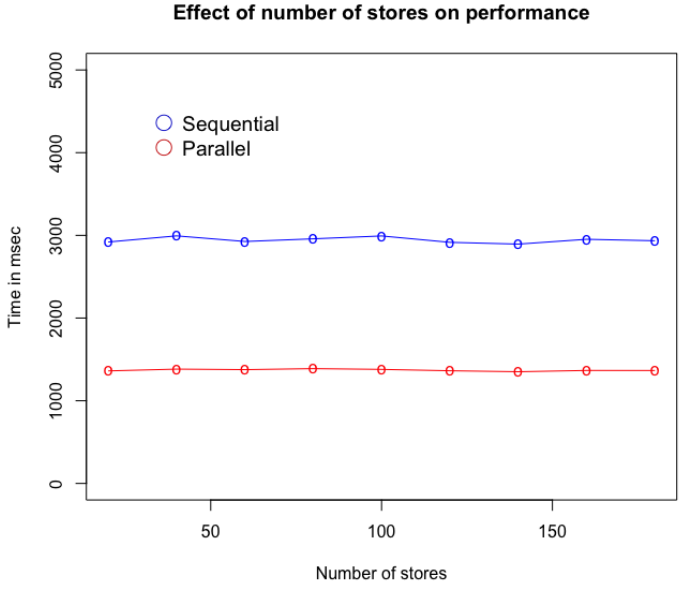
be more concurrent writes racing to reduce the stock. With increasing shopping list sizes, more and more purchases will find inadequate stock. As a result, we have reduced write collisions and reduced retries.

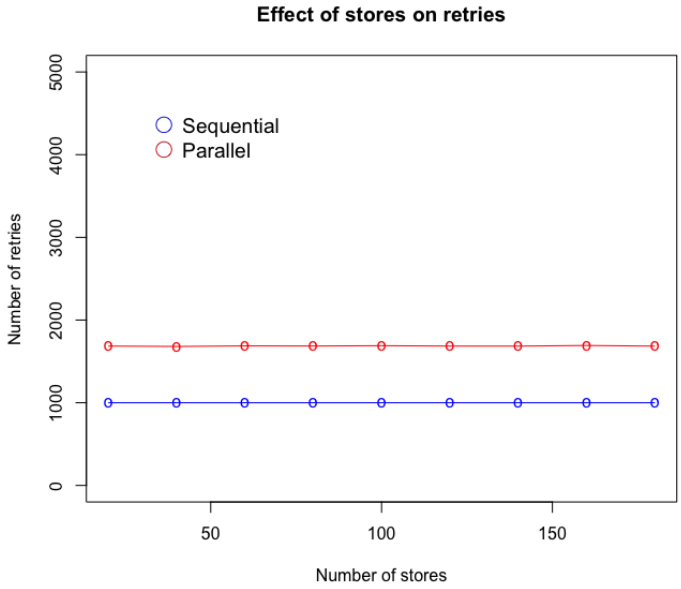


3.6 The effect of the number of stores

In this experiment, we vary the number of stores. There are two possible results of this experiment. First, increasing the number of stores increases the number of comparisons required to compute the cheapest store. Secondly, increasing stores increases product stock but not product choices. More stock increases latency since there is more stock to buy from. However, with a constant low number of customers and products, it is possible to see a constant latency and retry-count. This was the case for this experiment. Having a higher purchase rate (number of customers, per-product maximum, shopping list size) default would have shown an increasing latency and reducing retry count with an increase in stores.

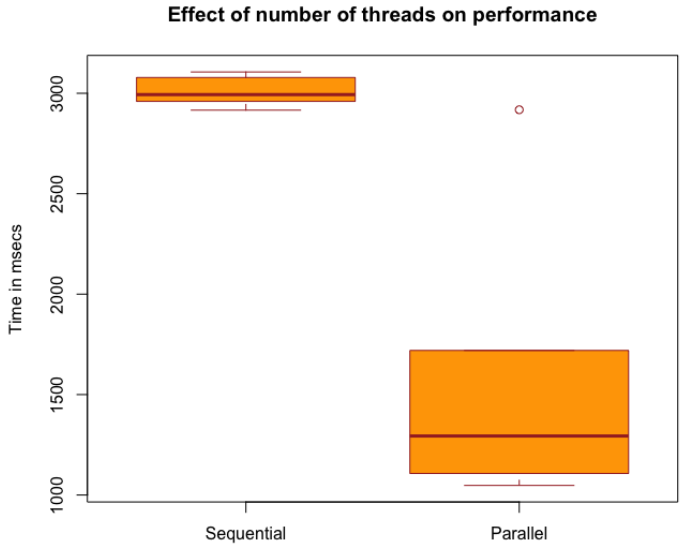

Effect of number of stores on performance
Time in msecs
3000
2500
2000
1500
Sequential
Parallel


Effect of number of stores on performance
Time in msec
5000
4000
3000
2000
1000
0
Sequential
Parallel
50
100
150
Number of stores

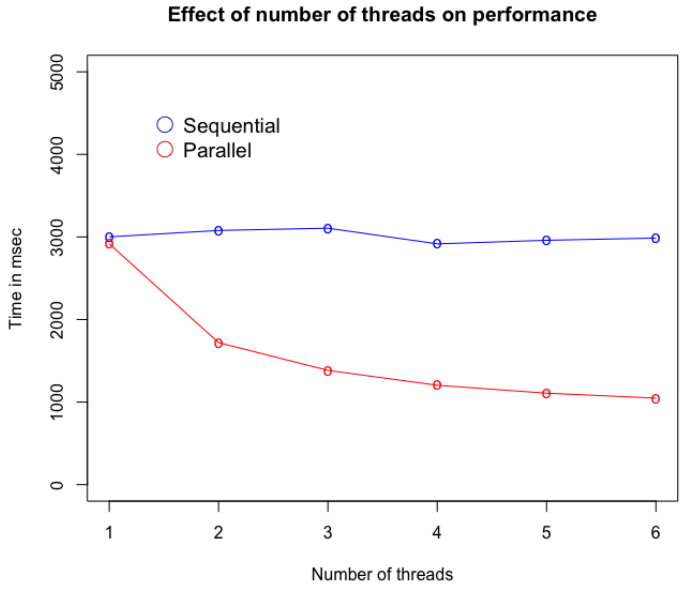


3.7 The effect of the number of threads

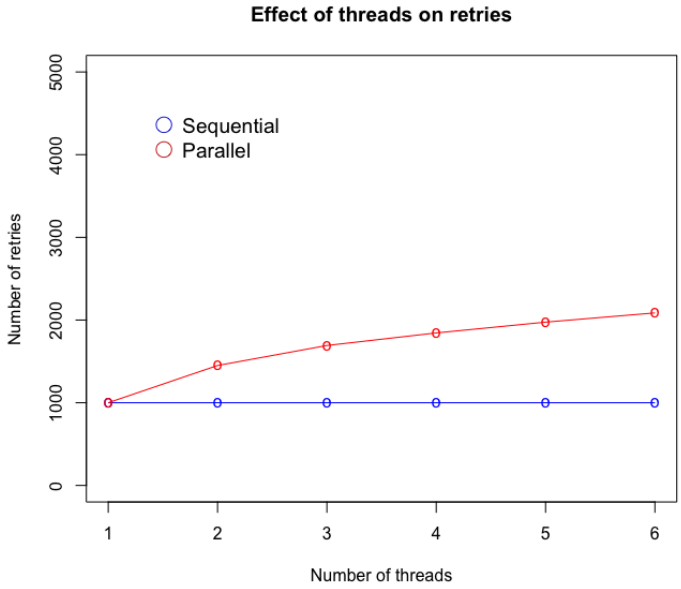
In this experiment, we increase the number of threads and run the sequential test with the default values. We see a significant difference between the mean latency of the parallel and sequential implementation with the default load. Moreover, we obtain speedup with increasing threads.



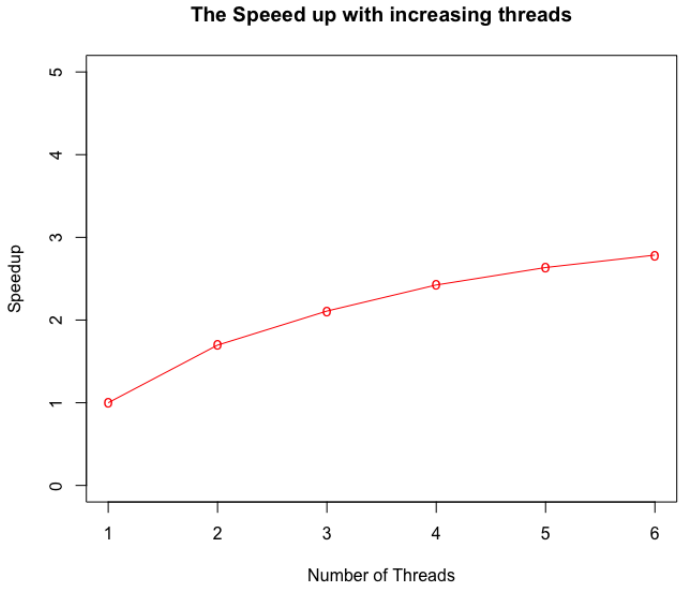
We see that with increasing threads, the latency of the parallel implementation reduces. The latency of the sequential implementation does not respond to increased threads as it processes one customer at a time.



With increasing threads, we record more transaction retries. This is because we have more threads racing to write the stock memory.

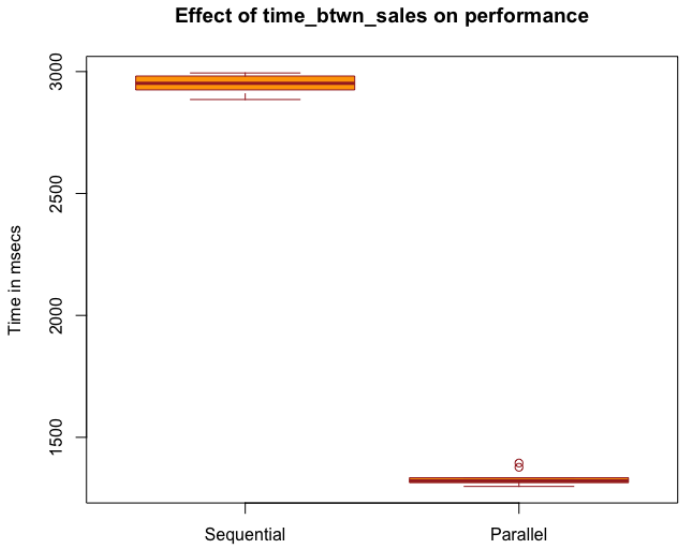


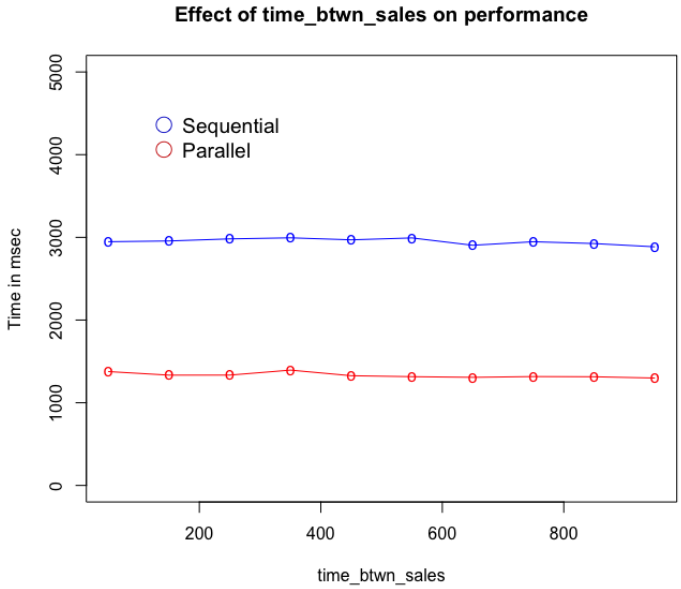
The speedup plot is a reflection of the latency curve. Note that the performance of one thread is equal to the performance of the sequential implementation.

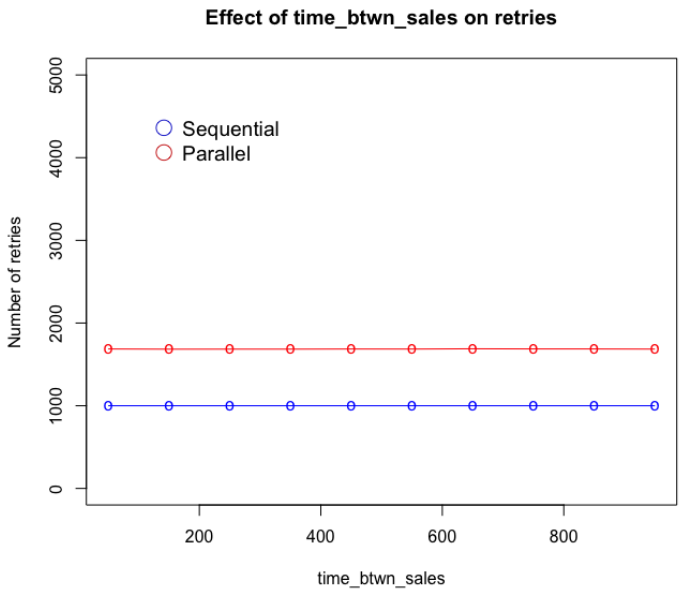


3.8 The effect of the time between sales

In this experiment, we vary the time between sales. This is a sleep period before the sales period. Since the sales process happens in a different thread, we do not observe latency change from the time between sales. Moreover, the time between sales affects prices, and we only read from that memory. Reads do not lead transaction retries, and neither do they block. Although we observe latency differences between the parallel and sequential implementations, we can decipher that they are caused by the other default settings of the experiment. The slopes of the retries and latency curves are not affected by this experiment.



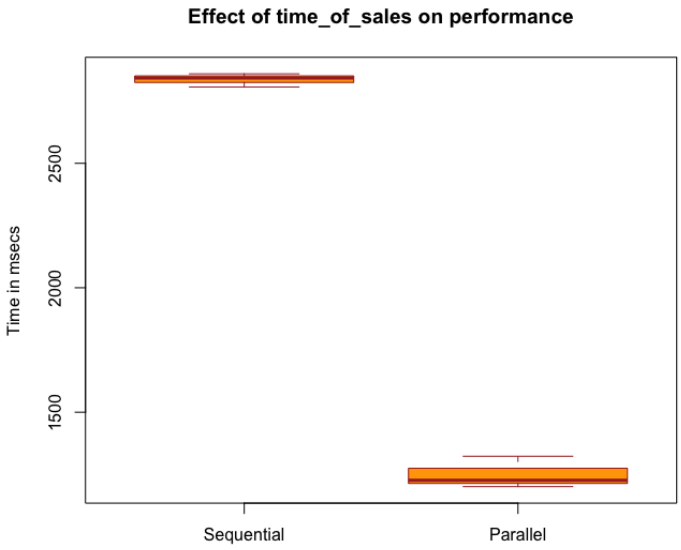


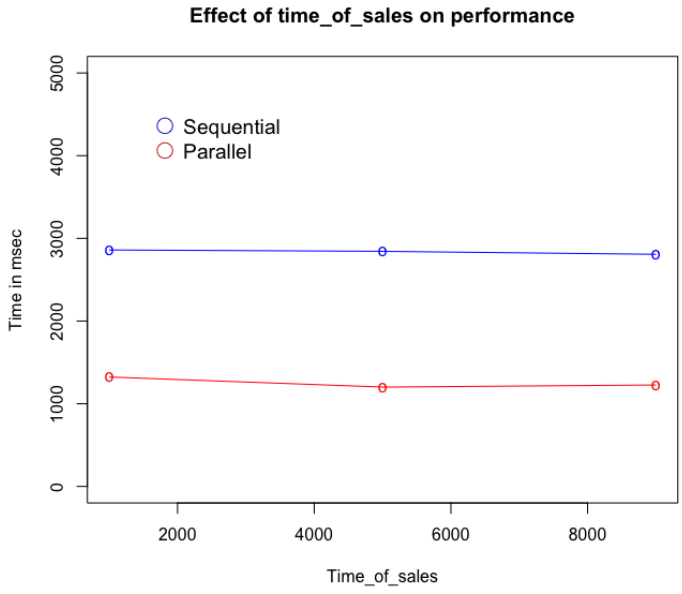


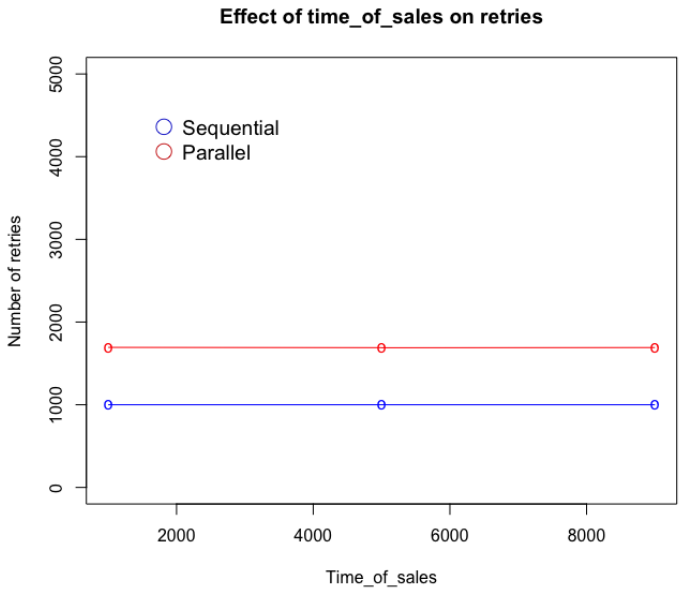
3.9 The effect of the time of sales

In this experiment, we vary the time of sales. This is the sleep period between discounting and restoring the original prices. Similarly, the sales process happens in a different thread, and we do not observe latency change from the

time of sales. The time of sales also affects prices which we only read from. This experiment does not change the slopes of the retries and latency curves.







4 Ensuring correctness

In the below test we ensure two things. Firstly, all quotations done for the sake of computing the cheapest store are either computed entirely before or after discounting. Secondly, we ensure that none of the stock after processing is in a negative balance.

```
(deftest priceTest []
    "We test that
    1. each customer quotation made for the sake of
    computing the cheapest store is either entirely done before or
    after discounts
    2. That after the sales process, none of the stocks is in a
        negative balance."
  ;the quotations is a list of the form [[storeid [[product-id,
      price]]]]
  (let [f1 (future (time (process-customers customers)))
        f2 (future (sales-process))]
       @f1
       @f2
        (await logger))
  (is
    (reducers/reduce  myand  true
      (doseq [storequatation @quatations]
        (map
          (fn [store-id product-price-list]
            (let [zero_or_one
              (map (fn [product-id price]
                (let [marked_price (get-price store-id product-id)]
```

```
                  (if (= price marked_price) 1 0))
                product-price-list))]
              (or (= zero_or_one 0) (= zero_or_one 1)))
            storequatation)))))
  (is
      (reducers/reduce myand true
          (map
              (fn [lst] (reducers/reduce myand true
                  (map notzero lst)))
              (map-deref stock))
          )))
```

5 Insight Questions

5.1 What would be the effect of excluding the sales process from this implementation?

The sales process runs inject a sleep period a different thread. The sales process does not affect customer processing. It affects the reads and writes of the prices. We only read from prices and reads do not block.

5.2 An alternative parallel implementation

We would use atoms to lock the stock instead of refs. Because we cannot coordinate atoms, we do not have enough freedom to lock each stock item and ensure correctness. According to the problem description, we can lock each store's entire stock product line stock into a single atom. This will guarantee the correctness and some degree of flexibility albeit lower than that offered by the finer granularity of coordinated refs in a single transaction.