Lab1 PDP - Inventory Store

Andrei Bratu 932

Purpose

The purpose of the project is to assess the benefits of parallelism in a non-cooperative environment i.e. the threads are using shared data, which must be locked in order to prevent data races. For this benchmark, a mock inventory management application is used.

Code Overview

The domain package of the problem contains **Product** and **Order** class, which describe the problem domain. An **Order** consists of multiple Products and the quantity they were ordered in. Of interest is the **command** sub-package, which contains tasks to be run by the threads. They leverage the common **InventoryRepository** interface. The commands are:

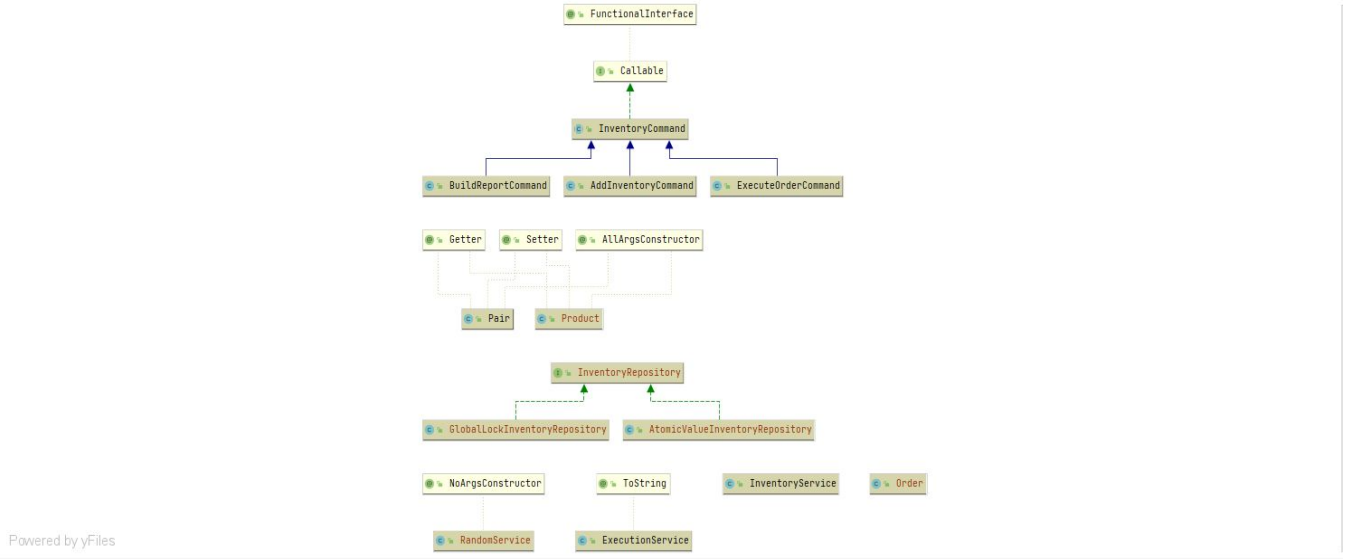
- **AddInventoryCommand** - initial filling in of the inventory
- **ExecuteOrderCommand** - execute a transaction which results in inventory subtraction and increase in profits
- **BuildReportCommand** - report on completed orders

The repository package is the center point of the experiment framework. InventoryRepository interface must be followed to add new parallelism experiments. Current repositories are:

- **GlobalLockInventoryRepository** holds a HashMap for **inventory** and a List for executed transactions - **executedOrders**. This repository uses a global lock on both **inventory** and **executedOrders**.
- **AtomicValueInventoryRepository** follows a similar approach with the previous experiment, exception being the fact that the HashMap uses **AtomicIntegers** in order to track **Product** quantity. Thus, it is not necessary to lock the entire HashMap when modifying quantities, and a lock is necessary only when adding completed **Orders** to **exectuedOrders**.

The service package contains three services:

- **RandomService** for trivial tasks like **generateRandomInt** or **generateRandomString**
- **InventoryService** is tasked with generating a random **Products** inventory and building a tasks schedule to be used in experiments. **InventoryService** generates and splits the random inventory into chunks, shuffles them, and groups all of them under transactions. When chunks are integrated inside Different quantities of possibly multiple products are found in any given transaction, and transactions sum up to the entire inventory.
- **ExecutionService** is responsible for benchmarking parallelism experiments. It takes in an **InventoryRepository** and a schedule of **InventoryCommand** callables that are the schedule to be used as a benchmark. An ExecutorService is used to manage the threads.



Methodology and Results

InventoryService leverages **RandomService** to generate distinct products, with quantities between 5000 and 50000.

CPU architecture, number of logical cores (i.e. including hyper-threads), used threads, average run time and number of trials are the collected metrics. Each experiment runs for 50 trials in order to obtain significant results for the run time metric.

Table below lists the results. Number of trials, architecture, and logical number of cores are listed in a second table, since they are constant.

Repository	Threads	Distinct Products	Run Time
GlobalLockInventoryRepository	1	100	0.6312
GlobalLockInventoryRepository	3	100	0.7167
GlobalLockInventoryRepository	5	100	0.7417
GlobalLockInventoryRepository	10	100	0.8371
GlobalLockInventoryRepository	20	100	0.8442
AtomicValueInventoryRepository	1	100	0.7328
AtomicValueInventoryRepository	3	100	0.7425

AtomicValueInventoryRepository	5	100	0.7663
AtomicValueInventoryRepository	10	100	0.7358
AtomicValueInventoryRepository	20	100	0.72426
GlobalLockInventoryRepository	1	500	17.92636
GlobalLockInventoryRepository	3	500	17.62734
GlobalLockInventoryRepository	5	500	19.11115
GlobalLockInventoryRepository	10	500	19.02839
GlobalLockInventoryRepository	20	500	21.08598
AtomicValueInventoryRepository	1	500	17.89814
AtomicValueInventoryRepository	3	500	19.44796
AtomicValueInventoryRepository	5	500	19.86616
AtomicValueInventoryRepository	10	500	18.84094
AtomicValueInventoryRepository	20	500	17.911179

Number Trials	Architecture	Logical Number Cores
50	amd64	8

Conclusions

In a non-cooperative environment, creating and managing threads and locks brings minor improvements at best but usually adds a noticeable overhead. The overhead tends to be proportional with the number of threads, and becomes more pronounced as the number of spawned threads becomes larger than the number of logical cores.