Mikhail Mitkevich

R/C++ developer

January 15, 2019

Outline

- *1994* Learned C++ by Stroustrup, aged 15
- *1996-2002* Msc in applied mathematics, Moscow institute of physics and technology (MIPT.ru)
- *2002-2006* C++/win32 developer, custom DSP chip gcc toolchain porting, ispras.ru
- *2006-2008* C++/C# developer, reinforcement learning applied to portfolio optimization, ccas.ru
- *2008-2016* java/linux developer, realtime distributed call center software, java/linux/oracle, Alcatel-Lucent
- *2016...* R/C++ and java developer in a proprietary hedge fund, both research and low latency execution roles

Programming

OS

- Ubuntu/CentOS
- OSX
- Windows NT (in the past)

Languages

- C++ 17
- java 8
- R, python
- javascript/nodejs
- C# (in the past)

Databases

- postgreSQL
- clickhouse
- lmdb, libsqlite
- oracle, mysql (in the past)

Tools and libraries

Tools

- cmake, make
- gdb, varlgrind
- conan, bintray.com
- gradle, maven
- git, gitlab, github

Libraries

- C++: stl, boost {asio, beast, thread, filesystem}, Rcpp, rxcpp
- R : dplyr, ggplot, purrr, shiny
- python: pandas, numpy, asyncio

My code: Rticks backtesting library (C++)

- gamma strategy (mean reversion): https://github.com/mmitkevich/rticks_v1/blob/master/src/gamma.h
- functional reactive streams design inspired by Reactive Extensions library (reactivex.io) http://rxmarbles.com, http://reactivex.io, http://github.com/ReactiveX/RxCpp
- utilizes C++ 14 and templates to produce optimized code using higher-level concepts
- Goals: to became a realtime stream-based alternative to R's dplyr, suitable for backtest/optimization and low-latency exectuion

Use case: crude marketmaking model 1/2

Suppose that:

- We want to be market makers of closest quarterly Crude Oil futures contracts on Comex and Nymex exchanges
- And keep our portfolio delta $\Delta(t)$ (e.g. normalized net futures position expressed in dollars) small enough to keep risks small:

$$\Delta(t) = [\sum_{i \in \{CL.1@COMEX, CL.1@NYMEX\}} \Delta_i(t)] \rightarrow \min_{STRATEGY.PARAMETERS}$$

- To start with, we choose our bid price S_{BID} and ask price S_{ASK} at fixed distance $D_{HALF.SPREAD}$ from some fair price $S_{FAIR.PRICE}$, defined later:

$$S_{BID,ASK} = S_{FAIR.PRICE} \mp D_{HALF.SPREAD}$$

- For simplicity we treat both futures essentially the same so we could mix their order books into single order book and use midprice as fair price:

$$S_{FAIR.PRICE} = 0.5(S_{BEST.BID} + S_{BEST.ASK})$$

Use case: crude marketmaking model 2/2

- We choose our buy order quantity Γ_{BUY} and (negative) sell order quantity Γ_{SELL} to keep our Delta small, but continue to make markets:

$$\Gamma^i_{BUY}(t) = \Gamma_0 + \max(0, -\Delta)$$

$$\Gamma^i_{SELL}(t) = -(\Gamma_0 + \max(0, \Delta))$$

- When executed, these orders would bring our Δ closer to zero

Concept: Functional strategy DSL 0

- In the following couple of slides I describe preliminary design of functional DSL language aimed for R & python quants to formally specify high frequency trading strategies for backtesting purposes
- Strategies are formulated as functional transformations of data and execution feeds (streams) so that the same code could be used in production as well

Concept: Market data feeds 1

First, note that conceptually all level1, level2, level3 and execution report feeds generalize to streams of tuples

```
feed(t) = [timestamp, symbol, op, price, qty]
```

Where operation *op* could have following values:

- PLACED: order was inserted into the order book
- CANCELED: order was removed from the order book
- FILLED: order was fully filled and is out of the book
- PART_FILLED: order filled partially and is still in the book
- UPDATED: cumulative quantity at order book price level has changed
- RESET, SNAPSHOT: these two could be used to synchronize order book snapshots with server

We leave transactions (PLACE, AMEND, CANCEL) out of scope for now.

Concept: streams DSL inspired by GNU R's dplyr package 2

dplyr is a popular GNU R package for data.frame manipulations.

Let's imagine dplyr-like DSL to define our trading strategy. Let's start with defining the feeds using hypothetical *data_streams* function

```
symbols = c("CL.1@COMEX", "CL.1@NYMEX")
s0 = data_streams(
  level2=level2_feed(symbols),
  execs=exec_feed("mike@GS","peter@DB"))
```

So printing feeds could give us the following output

```
>tail(feeds)
level2|UPDATE|17:23:14.323|CL.1@COMEX | 65.50 | 100
level2|UPDATE|17:23:14.324|CL.1@NYMEX | 65.52 | -100
execs |FILL  |17:23:14.324|CL.1@COMEX | 65.51 | 50 | mik
```

Concept: stream mutation in functional style 3

Actual position could be calculating by summing *qty* for all *executions*:

```
s1 = s0 %>% mutate(
   position=cumsum(executions))
```

Note, that:

- summation is done for each symbol individually.
- logically position stream is also tuple [price, qty] so we should define arithmetic operations on such tuples

Concept: simplify arithmetics

Let's denote

- S:price (in USD),
- Q:qty (in items),
- $X = [S, Q]$:vector of both,
- $U(X) = SQ$: value of this vector

Let's define plus, minus, mul, div operators

$$X_1 + X_2 = [S_1, Q_1] + [S_2, Q_2] = [S_1 Q_1 + S_2 Q_2, 1] \rightarrow U = S_1 Q_1 + S_2 Q_2 = U_1 + U_2$$

$$X_1 - X_2 = [S_1 Q_1 - S_2 Q_2, 1] \rightarrow U = U_1 - U_2$$

$$X_1 * X_2 = [S_1 Q_1 S_2 Q_2, 1] \rightarrow U = U_1 U_2$$

$$X_1 / X_2 = [S_1 Q_1 / S_2 Q_2, 1] \rightarrow U = U_1 / U_2$$

This corresponds to casting [price, qty] tuple to price*qty real number, and casting back real number to [number, 1] tuple. This would simplify arithmetics

Concept: mixing order books 4

To mix order books we transform *level*2 feed to have common symbol and then use *level*2_*to*_*level*1 function which

- takes *level*2 stream as its input
- reconstructs order book state on it (filtering matched price levels)
- outputs required *level*1 feed (which basically contains only SNAPSHOT records):

Concept: calculating required values 5

Also we transform last *position* and *mid* price into delta.

```
contract_lot=10 # barrels
s2 = s1 %>% mutate(
  level2=level2%>%mutate(symbol='CL.1@MIXED'),
  level1=level2_to_level1(level2),
  best.bid=level1 %>% filter(qty>0),
  best.ask=level1 %>% filter(qty<0),
  fair.price=0.5*(best.bid+best.ask),
  delta=position*fair.price*contract_lot)
```

Note, that stream of position is a single number,

Concept: reducing over portfolio 6

In the previous example we have calculated delta for each contract, so now we need to reduce it over all symbols in the portfolio

```
s3 = s2 %>% mutate(
  Delta=reduce(delta, ~ .x+.y)
)
```

Now our $s3$ stream contains everything we need to calculate fair price and our buy/sell quotes:

- Delta=Δ: total delta of the portfolio, in dollars
- best.bid=S_{BID}, best.ask=S_{ASK}, fair.price=$S_{FAIR.PRICE}$: mixed book bid, ask and fair price calculated as their average

Concept: calculating final market maker quotes 7

```
gamma0=100 # USD
half_spread=0.01 # USD
s4 = s3 %>% mutate(
    buy = (fair.price-half_spread) %>% mutate(
          qty=(max(0,-Delta)+gamma0)
                  /contract_lot/price),
    sell = (fair.price+half_spread) %>% mutate(
          qty=-(max(0,Delta)+gamma0)
                 /contract_lot/price))
```

These buy and ask quotes are subject to be maintained in the market by high-frequency execution service. This service receives desired buy and sell quotes (with respective quantities) and sends orders accordingly

Concept: Execution, further improvements 8

- Possible latency reduction could be achieved by mixed software/hardware design. For example, our simple strategy only uses simple arithmetics, max/min operations and orderbook mixing operations. If properly implemented in FPGA hardware it could be possible to reduce latency introduced by software.
- DSL language could be implemented not only with R dplyr-like syntax. Actually all 'mutate' clauses are really optional. Usability by quants is the main concern.

My code: offheap order matching engine (Java)

- https://github.com/mmitkevich/lob-java/blob/master/src/main/java/org/freeticks/lob/OffHeapBook.java
- uses DirectBuffer to manipulate offheap memory and minimize allocations
- price level index implemented as plain array instead of sorted rb-tree
- orders are allocated in offheap circular arena

My code: lockless SPSC queue (.NET)

- https://github.com/mmitkevich/Snail/blob/master/Snail/Threading/BQueue.cs
- CLR implementation of bounded lockless queue addressing false sharing problem for low latency core-to-core communication

Links

https://linkedin.com/in/mmitkevich

https://github.com/mmitkevich