A Report from the Real World

Lennart Augustsson

Standard Chartered Bank

May 7, 2012

Introduction

Introduction

Financial computing

Introduction

Financial computing

Functional Programming at SCB

Introduction

Financial computing

Functional Programming at SCB

Parallel FP at SCB

Standard Chartered Bank

- Operates mainly in Asia, Africa, Middle East
- Headquarters in London
- 70 countries in total
- Employs 87,000 people
- Fourth largest bank in Europe

Me

- MSc, PhD from Chalmers
- Lecturer at Chalmers
- Consultant at CR&T
- Hardware at Sandburst
- Banking at CS & SCB

Me

- MSc, PhD from Chalmers
- Lecturer at Chalmers
- Consultant at CR&T
- Hardware at Sandburst
- Banking at CS & SCB
- But mostly, I write compilers

Parallel FP at SCB

Parallel FP at SCB

```
pmap :: Strategy -> (a -> b) -> [a] -> [b]
```

How do (investment) banks us computers?

How do (investment) banks us computers?

- Compute price of products

How do (investment) banks us computers?

- Compute price of products
- Compute P&L (profit and loss) of current position

How do (investment) banks us computers?

- Compute price of products
- Compute P&L (profit and loss) of current position
- **Compute risk of current position**

What is a financial product?

Contractual obligation with a counter-party.

What is a financial product?

Contractual obligation with a counter-party.

Example

From 2012-01-01 you will pay me $100 every month for 12 months. At 2012-06-01 you will make a choice to get 2 Apple shares or 60 Cisco shares at 2013-01-01.

What is a financial product?

Contractual obligation with a counter-party.

Example

From 2012-01-01 you will pay me $100 every month for 12 months. At 2012-06-01 you will make a choice to get 2 Apple shares or 60 Cisco shares at 2013-01-01.

What is it worth to hold such a contract?

Contract

The same contract expressed in our DSL (mostly taken from Simon Peyton Jones and Jean-Marc Eber):

```
example =
  and (monthly 12 (2012-01-01) $
           recieve 100 USD)
      (give (at (2012-06-01) $
               or (at (2013-01-01) $ recieve 2  Apple)
                  (at (2013-01-01) $ recieve 60 Cisco)))
```

Pricing financial products

Pricing financial products

- Very simple products, e.g. options, can be priced analytically.

Pricing financial products

- Very simple products, e.g. options, can be priced analytically.
- Black-Scholes option pricing

  $$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

  Has solution

  $$C(S,t) = N(d_1)\,S - N(d_2)\,Ke^{-r(T-t)}$$

  $$d_1 = \frac{\ln(\frac{S}{K}) + (r + \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}}$$

  $$d_2 = \frac{\ln(\frac{S}{K}) + (r - \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}} = d_1 - \sigma\sqrt{T-t}$$

Pricing financial products

- Very simple products, e.g. options, can be priced analytically.
- Black-Scholes option pricing

  $$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

  Has solution

  $$C(S,t) = N(d_1)\, S - N(d_2)\, Ke^{-r(T-t)}$$

  $$d_1 = \frac{\ln(\frac{S}{K}) + (r + \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}}$$

  $$d_2 = \frac{\ln(\frac{S}{K}) + (r - \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}} = d_1 - \sigma\sqrt{T-t}$$
- Most products have to be priced using approxmate methods

Pricing financial products

- Very simple products, e.g. options, can be priced analytically.
- Black-Scholes option pricing

  $$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

  Has solution

  $$C(S,t) = N(d_1)\,S - N(d_2)\,Ke^{-r(T-t)}$$

  $$d_1 = \frac{\ln(\frac{S}{K}) + (r + \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}}$$

  $$d_2 = \frac{\ln(\frac{S}{K}) + (r - \frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}} = d_1 - \sigma\sqrt{T-t}$$

- Most products have to be priced using approxmate methods
-
  - Numerical solutions to PDEs (Partial Differential Equations), akin to the Laplace heat equation
  - Simulation using Monte-Carlo

Embarrassingly parallel

Embarrassingly parallel

- Monte-Carlo is just simulating the movement of various financial instruments (interest rates, stock prices, etc) and computing a final value. Average over a large number of Monte-Carlo runs.

Embarrassingly parallel

- Monte-Carlo is just simulating the movement of various financial instruments (interest rates, stock prices, etc) and computing a final value. Average over a large number of Monte-Carlo runs.
- Computing risk positions is taking the derivatives of various inputs. This is usually done numerically.

Embarrassingly parallel

- Monte-Carlo is just simulating the movement of various financial instruments (interest rates, stock prices, etc) and computing a final value. Average over a large number of Monte-Carlo runs.
- Computing risk positions is taking the derivatives of various inputs. This is usually done numerically.
- Both of these have a lot of parallel independent computations, with just a little post-processing.

Embarrassingly parallel

- Monte-Carlo is just simulating the movement of various financial instruments (interest rates, stock prices, etc) and computing a final value. Average over a large number of Monte-Carlo runs.
- Computing risk positions is taking the derivatives of various inputs. This is usually done numerically.
- Both of these have a lot of parallel independent computations, with just a little post-processing.
- In short, lots of independent relatively large computations.

Other parallelism

Other parallelism

- High Frequency Trading
  - Automated trading with very low latency ($<$ 1ms)
  - Accounts for most trading these days

Functional Programming at SCB

Functional Programming at SCB

- Quant library, Cortex, used for pricing and risk.

Functional Programming at SCB

- Quant library, Cortex, used for pricing and risk.
- Low level numeric code written is C++.

Functional Programming at SCB

- Quant library, Cortex, used for pricing and risk.
- Low level numeric code written is C++.
- High level programming done in Mu, a strict dialect of Haskell.

Functional Programming at SCB

- Quant library, Cortex, used for pricing and risk.
- Low level numeric code written is C++.
- High level programming done in Mu, a strict dialect of Haskell.
- Callable from Mu, Haskell, C++, C#, Java, and Excel.

Functional Programming at SCB

- Quant library, Cortex, used for pricing and risk.
- Low level numeric code written is C++.
- High level programming done in Mu, a strict dialect of Haskell.
- Callable from Mu, Haskell, C++, C#, Java, and Excel.
- The purity of Haskell is essential!

Functional Programming at SCB

- Quant library, Cortex, used for pricing and risk.
- Low level numeric code written is C++.
- High level programming done in Mu, a strict dialect of Haskell.
- Callable from Mu, Haskell, C++, C#, Java, and Excel.
- The purity of Haskell is essential!
- (We hire Haskell programmers.)

FP Parallelism at SCB

```
pmap :: Strategy -> (a -> b) -> [a] -> [b]
```

Strategy

Strategy

- Sequential

```
sequential :: Strategy
```

Strategy

- Sequential

```
sequential :: Strategy
```

- Threaded, multiple threads in same process

```
threaded :: Int -> Strategy
```

Strategy

- Sequential
  `sequential :: Strategy`
- Threaded, multiple threads in same process
  `threaded :: Int -> Strategy`
- Process, multiple processes on the same computer
  `process :: Int -> Strategy`

Strategy

- Sequential

  `sequential :: Strategy`

- Threaded, multiple threads in same process

  `threaded :: Int -> Strategy`

- Process, multiple processes on the same computer

  `process :: Int -> Strategy`

- Nesting

  `nest :: Strategy -> Strategy -> Strategy`

Strategy

- Sequential
  ```
  sequential :: Strategy
  ```
- Threaded, multiple threads in same process
  ```
  threaded :: Int -> Strategy
  ```
- Process, multiple processes on the same computer
  ```
  process :: Int -> Strategy
  ```
- Nesting
  ```
  nest :: Strategy -> Strategy -> Strategy
  ```
- Grid
  ```
  grid :: GridName -> Int -> Strategy
  ```

Strategy, examples

Strategy, examples

- No parallelism

```
pmap sequential = map
```

Strategy, examples

- No parallelism

  `pmap sequential = map`

- Using 4 cores in a single process

  `pmap (threaded 4)`

Strategy, examples

- No parallelism
  `pmap sequential = map`
- Using 4 cores in a single process
  `pmap (threaded 4)`
- Use 4 cores in 4 processes
  `pmap (process 4)`

Strategy, examples

- No parallelism

```
pmap sequential = map
```

- Using 4 cores in a single process

```
pmap (threaded 4)
```

- Use 4 cores in 4 processes

```
pmap (process 4)
```

- Use 4 cores in 2 processes

```
pmap (nest (process 2) (threaded 2))
```

Strategy, examples

- No parallelism

  `pmap sequential = map`
- Using 4 cores in a single process

  `pmap (threaded 4)`
- Use 4 cores in 4 processes

  `pmap (process 4)`
- Use 4 cores in 2 processes

  `pmap (nest (process 2) (threaded 2))`
- Use 100 compute engines in the London test grid

  `pmap (grid "LDNtest" 100)`

Strategy, examples

- No parallelism

  `pmap sequential = map`

- Using 4 cores in a single process

  `pmap (threaded 4)`

- Use 4 cores in 4 processes

  `pmap (process 4)`

- Use 4 cores in 2 processes

  `pmap (nest (process 2) (threaded 2))`

- Use 100 compute engines in the London test grid

  `pmap (grid "LDNtest" 100)`

- Use 4096 cores in Kuala Lumpur production grid

  ```
  pmap (nest (grid "KLprod" 512) (nest (process 2)
  (threaded 4)))
  ```

Some more map functions

Some more map functions

- With IO, mapM for IO monad

```
pmapIO ::  Strategy -> (a -> IO b) -> [a] -> IO [b]
```

Some more map functions

- With IO, mapM for IO monad

```
pmapIO ::  Strategy -> (a -> IO b) -> [a] -> IO [b]
```

- With input only, mapM for SafeIO monad

```
pmapSafeIO ::  Strategy -> (a -> SafeIO b) -> [a] ->
SafeIO [b]
```

Some more map functions

- With IO, mapM for IO monad
  `pmapIO :: Strategy -> (a -> IO b) -> [a] -> IO [b]`
- With input only, mapM for SafeIO monad
  `pmapSafeIO :: Strategy -> (a -> SafeIO b) -> [a] -> SafeIO [b]`
- These function are not available on the grid. The grid cannot do IO.

Some more map functions

- With IO, mapM for IO monad

  `pmapIO :: Strategy -> (a -> IO b) -> [a] -> IO [b]`
- With input only, mapM for SafeIO monad

  `pmapSafeIO :: Strategy -> (a -> SafeIO b) -> [a] -> SafeIO [b]`
- These function are not available on the grid. The grid cannot do IO.
- The type system is crucial to know when something does IO.

Implementation implications

Implementation implications

- Arbitrary values (including functions) need to be transferred between machines.

Implementation implications

- Arbitrary values (including functions) need to be transferred between machines.
- The machines may not even have the same architecture.

Implementation implications

- Arbitrary values (including functions) need to be transferred between machines.
- The machines may not even have the same architecture.
- Serializing arbitrary values cannot be done at the Haskell level.
  - Need to preserve unobserval properties like cycles.
  - Serializing function between architectures precludes sending machine code.

Implementation implications

- Arbitrary values (including functions) need to be transferred between machines.
- The machines may not even have the same architecture.
- Serializing arbitrary values cannot be done at the Haskell level.
  - Need to preserve unobserval properties like cycles.
  - Serializing function between architectures precludes sending machine code.
- Other languages with serialization
  - Erlang
  - Clean
  - (Java)

Serializing data

Serializing data

- Every basic data structures knows how to convert itself to/from a bytestream.

Serializing data

- Every basic data structures knows how to convert itself to/from a bytestream.
- Serialization memoized to make sure each object in memory is only transferred once.

Serializing data

- Every basic data structures knows how to convert itself to/from a bytestream.
- Serialization memoized to make sure each object in memory is only transferred once.
- Some objects are tricky, like open network connections.

Serializing functions

Serializing functions

- Functions can be pure code, or partial applications.
  - Partial applications (closures) is just pure code and a tuple of values.

Serializing functions

- Functions can be pure code, or partial applications.
  - Partial applications (closures) is just pure code and a tuple of values.
- Pure functions are stored and serialized as byte code.

Serializing functions

- Functions can be pure code, or partial applications.
  - Partial applications (closures) is just pure code and a tuple of values.
- Pure functions are stored and serialized as byte code.
- For machine code the bytecode is JITed using LLVM.

Serializing functions

- Functions can be pure code, or partial applications.
  - Partial applications (closures) is just pure code and a tuple of values.
- Pure functions are stored and serialized as byte code.
- For machine code the bytecode is JITed using LLVM.
- For serialization, send the bytecode, and re-JIT at the destination.

Real world complications, versions

Real world complications, versions

- People will serialize and save data.
  - Must be able to read old data forever.
  - Backwards compatibility introduces a lot complications and code bloat.

Real world complications, versions

- People will serialize and save data.
  - Must be able to read old data forever.
  - Backwards compatibility introduces a lot complications and code bloat.
- The grid is often running an older version of the software.
  - New versions of data structures must be introduced in stages.

Concurrency

When building user interfaces concurrency is very useful; it also has some amount of parallelism.

Conclusions

- A lot of parallelism is very easy to find.
- A pure language is huge advantage.
- But utilizing parallelism still hard for practical reasons.