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
Introduction

Financial computing

Functional Programming at SCB

Parallel FP at SCB

Conclusions
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
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
pmap :: Strategy -> (a -> b) -> [a] -> [b]
How do (investment) banks use computers?

- Compute price of products
- Compute P&L (profit and loss) of current position
- Compute risk of current position
How do (investment) banks use computers?

- Compute price of products
- Compute P&L (profit and loss) of current position
- Compute risk of current position
How do (investment) banks use computers?

- Compute price of products
- Compute P&L (profit and loss) of current position
How do (investment) banks use 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.

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

```plaintext
def 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

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

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

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

\[
d_2 = d_1 - \sigma \sqrt{T-t}
\]

Most products have to be priced using approximate methods

Numerical solutions to PDEs (Partial Differential Equations), akin to the Laplace heat equation

Simulation using Monte-Carlo
Very simple products, e.g. options, can be priced analytically.

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

Has solution

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

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

\[
d_2 = d_1 - \sigma \sqrt{T-t}
\]

Most products have to be priced using approximate methods

Numerical solutions to PDEs (Partial Differential Equations), akin to
the Laplace heat equation

Simulation using Monte-Carlo
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
  \[ C(S, t) = N(d_1) S - N(d_2) Ke^{-r(T-t)} \]
  \[ d_1 = \frac{\ln(S/K) + (r + \frac{\sigma^2}{2})(T-t)}{\sigma \sqrt{T-t}} \]
  \[ d_2 = \frac{\ln(S/K) + (r - \frac{\sigma^2}{2})(T-t)}{\sigma \sqrt{T-t}} = d_1 - \sigma \sqrt{T-t} \]
- Most products have to be priced using approximate 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

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

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

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

- Most products have to be priced using approximate methods

  - Numerical solutions to PDEs (Partial Differential Equations), akin to the Laplace heat equation
  - Simulation using Monte-Carlo
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.
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.
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

High Frequency Trading

Automated trading with very low latency (<1ms)

Accounts for most trading these days
Other parallelism

- High Frequency Trading
  - Automated trading with very low latency (< 1ms)
  - Accounts for most trading these days
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.)
Functional Programming at SCB

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

(We hire Haskell programmers.)
Functional Programming at SCB

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

(We hire Haskell programmers.)
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.
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.
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!
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.)
pmap :: Strategy -> (a -> b) -> [a] -> [b]
Strategy

Sequential
- Strategy
- Threaded, multiple threads in same process

Process
- Strategy
- Int >

Nesting
- Strategy
- Nest
- Strategy

Grid
- GridName
- Int
- Strategy
Strategy

- Sequential
  sequential :: Strategy
- Sequential
  sequential :: Strategy

- Threaded, multiple threads in same process
  threaded :: Int -> Strategy
Sequential

sequential :: Strategy

Threaded, multiple threads in same process

threaded :: Int -> Strategy

Process, multiple processes on the same computer

process :: Int -> Strategy
- Sequential
  \( \text{sequential} :: \text{Strategy} \)

- Threaded, multiple threads in same process
  \( \text{threaded} :: \text{Int} \rightarrow \text{Strategy} \)

- Process, multiple processes on the same computer
  \( \text{process} :: \text{Int} \rightarrow \text{Strategy} \)

- Nesting
  \( \text{nest} :: \text{Strategy} \rightarrow \text{Strategy} \rightarrow \text{Strategy} \)
Strategy

- **Sequential**
  
  \[
  \text{sequential} :: \text{Strategy}
  \]

- **Threaded, multiple threads in same process**
  
  \[
  \text{threaded} :: \text{Int} \to \text{Strategy}
  \]

- **Process, multiple processes on the same computer**
  
  \[
  \text{process} :: \text{Int} \to \text{Strategy}
  \]

- **Nesting**
  
  \[
  \text{nest} :: \text{Strategy} \to \text{Strategy} \to \text{Strategy}
  \]

- **Grid**
  
  \[
  \text{grid} :: \text{GridName} \to \text{Int} \to \text{Strategy}
  \]
Strategy, examples

No parallelism

- `pmap sequential = map`

Using 4 cores in a single process

- `pmap (threaded 4)`

Using 4 cores in 4 processes

- `pmap (process 4)`

Using 4 cores in 2 processes

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

Using 100 compute engines in the London test grid

- `pmap (grid "LDNtest" 100)`

Using 4096 cores in Kuala Lumpur production grid

- `pmap (nest (grid "KLprod" 512) (nest (process 2) (threaded 4)))`
Strategy, examples

- No parallelism
  pmap sequential = map
Strategy, examples

- No parallelism
  \[ \text{pmap sequential} = \text{map} \]

- Using 4 cores in a single process
  \[ \text{pmap (threaded 4)} \]
Strategy, examples

- No parallelism
  \[ \text{pmap sequential} = \text{map} \]

- Using 4 cores in a single process
  \[ \text{pmap (threaded 4)} \]

- Use 4 cores in 4 processes
  \[ \text{pmap (process 4)} \]
Strategy, examples

- No parallelism
  \[ \text{pmap sequential} = \text{map} \]

- Using 4 cores in a single process
  \[ \text{pmap (threaded 4)} \]

- Use 4 cores in 4 processes
  \[ \text{pmap (process 4)} \]

- Use 4 cores in 2 processes
  \[ \text{pmap (nest (process 2) (threaded 2))} \]

- Use 100 compute engines in the London test grid
  \[ \text{pmap (grid "LDNtest" 100)} \]

- Use 4096 cores in Kuala Lumpur production grid
  \[ \text{pmap (nest (grid "KLprod" 512) (nest (process 2) (threaded 4)))} \]
Strategy, examples

- No parallelism
  \[ \text{pmap} \text{ sequential} = \text{map} \]
- Using 4 cores in a single process
  \[ \text{pmap} \text{ (threaded 4)} \]
- Use 4 cores in 4 processes
  \[ \text{pmap} \text{ (process 4)} \]
- Use 4 cores in 2 processes
  \[ \text{pmap} \text{ (nest (process 2) (threaded 2))} \]
- Use 100 compute engines in the London test grid
  \[ \text{pmap} \text{ (grid "LDNtest" 100)} \]
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

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 functions are not available on the grid. The grid cannot do IO.

The type system is crucial to know when something does IO.
Some more map functions

- With IO, mapM for IO monad
  
  \[ pmapIO :: \text{Strategy} \rightarrow (a \rightarrow \text{IO} \ b) \rightarrow [a] \rightarrow \text{IO} \ [b] \]
Some more map functions

- With IO, mapM for IO monad
  \[
  \text{pmapIO :: Strategy} \rightarrow (a \rightarrow \text{IO } b) \rightarrow [a] \rightarrow \text{IO } [b]
  \]

- With input only, mapM for SafeIO monad
  \[
  \text{pmapSafeIO :: Strategy} \rightarrow (a \rightarrow \text{SafeIO } b) \rightarrow [a] \rightarrow \text{SafeIO } [b]
  \]
Some more map functions

- With IO, mapM for IO monad
  $$\text{pmapIO} :: \text{Strategy} \to (a \to \text{IO } b) \to [a] \to \text{IO } [b]$$

- With input only, mapM for SafeIO monad
  $$\text{pmapSafeIO} :: \text{Strategy} \to (a \to \text{SafeIO } b) \to [a] \to \text{SafeIO } [b]$$

- These function are not available on the grid. The grid cannot do IO.
Some more map functions

- With IO, mapM for IO monad
  \[ pmapI0 :: \text{Strategy} \to (a \to \text{IO } b) \to [a] \to \text{IO } [b] \]

- With input only, mapM for SafeIO monad
  \[ pmapSafeI0 :: \text{Strategy} \to (a \to \text{SafeIO } b) \to [a] \to \text{SafeIO } [b] \]

- 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

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 unobservational properties like cycles. Serializing function between architectures precludes sending machine code. Other languages with serialization include Erlang, Clean, and Java.
Arbitrary values (including functions) need to be transferred between machines.
- 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 unobservability 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 unobservable properties like cycles.
  - Serializing function between architectures precludes sending machine code.
- Other languages with serialization
  - Erlang
  - Clean
  - (Java)
Serializing data

Every basic data structure 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.
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.
Every basic data structure 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.
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.
Functions can be pure code, or partial applications.

- Partial applications (closures) is just pure code and a tuple of values.
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.
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

People will serialize and save data. Must be able to read old data forever. Backwards compatibility introduces a lot of 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.
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.
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.