Monte-Carlo based Pricing in Haskell

(sliced out of other presentations)

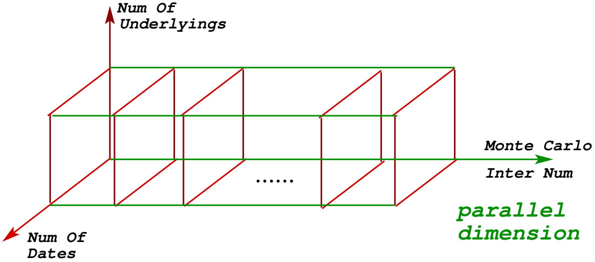
Jost Berthold and Cosmin Oancea

berthold@diku.dk and cosmin.oancea@diku.dk

Department of Computer Science
University of Copenhagen

May 7, 2012

Bird's Eye View



- Generate independent Sobol sequences (uniform in $[0, 1)$)
- Transform into Gaussian distribution $(-\infty, \infty)$ (numerical $\int$)
- Brownian bridge for all dates, for each underlying
- Compute trajectory using Black-Scholes
- Monte-Carlo aggregation with product's payoff and discount

Bird's Eye View

Haskell Code

```
mc_pricing n = let conf = init_sobol_orig n
                   gains  = map ( payoff          conf
                                . black_scholes   conf
                                . brownian_bridge conf
                                . gaussian
                                . sobolInd        conf ) [1..n]
                   price  = sum gains
                   factor = 1 / (fromIntegral N)
               in  factor * price
```

- Generate independent Sobol sequences (uniform in $[0, 1)$)
- Transform into Gaussian distribution $(-\infty, \infty)$ (numerical $\int$)
- Brownian bridge for all dates, for each underlying
- Compute trajectory using Black-Scholes
- Monte-Carlo aggregation with product's payoff and discount

Functional Basic Blocks

Haskell Code Inlined

```
mc_pricing n = let conf = init_sobol_orig n
                   gains  = map ( payoff          conf
                                . black_scholes   conf
                                . brownian_bridge conf
                                . gaussian
                                . sobolInd        conf ) [1..n]
                   price  = sum gains
                   factor = 1 / (fromIntegral N)
               in  factor * price
```

Functional Basic Blocks Types

```
sobolInd    :: Pricing_Data -> Index -> [Real]
-- sobolRec :: Pricing_Data -> [Elem] -> Index -> [Real]
gaussian    :: [Real] -> [Real]
--------------------------------------------------
brownian_bridge :: Pricing_Data ->  [Real]   -> [[Real]]
black_scholes   :: Pricing_Data -> [[Real]] -> [[Real]]
payoff          :: Pricing_Data -> [[Real]] ->   Real
```

$[0,1)^{u \cdot d}$

$[0,1)^{u \cdot d}$

$\mathbb{R}^{u \cdot d}$

$\mathbb{R}^{u \times d}$

$\mathbb{R}^{u \times d}$

$\mathbb{R}$

Computing a New Trajectory with Black-Scholes

Black-Scholes

```
black_scholes :: Pricing_Data -> [[SpecReal]] -> [[SpecReal]]
black_scholes c = mkPrices c . correlate_deltas c . mkDeltas
```

Compute deltas:

$$\delta_{i,j} = r_{i,j} - r_{i-1,j}\ \forall_{i>0},\ \ \delta_{0,j} = r_{0,j}$$

```
mkDeltas :: Num r => [[r]] -> [[r]]
mkDeltas rows@(row1:restrows) = row1 : zipWith (zipWith (-)) restrows rows
```

Correlate delta matrix using C matrix:

$$\delta^c_{i,j} = \sum_{k=0}^{j} \delta_{i,k} \cdot c_{j,k}$$

```
correlate_deltas :: Pricing_Data -> [[SpecReal]] -> [[SpecReal]]
correlate_deltas Pricing_Data{..} zds
    =  map (\zr -> map (sum . zipWith (*) zr) md_c) zds
```

Compute Prices with Black-Scholes:

$$s_{i,j} = s_{0,j} \cdot e^{\sum \delta^c_{k,j} \cdot \sigma_{k,j} + dr_{k,j}}$$

```
mkPrices :: Pricing_Data -> [[SpecReal]] -> [[SpecReal]]
mkPrices Pricing_Data{..} noises
  = let e_rows = map (map exp)  -- dc[l][j]*vol[l][j] + dr[l][j]
                 [ zipWith (+) (zipWith (*) n_row vol_row) dr_row
                  | (n_row, vol_row, dr_row) <- zip3 noises md_vols md_drifts ]
      in tail (scanl (zipWith (*)) md_starts e_rows)
```

Black-Scholes: C++

C++ Code

```
for (i = n_und * n_dates - 1; i >= n_und; i--) z[i] -= z[i-n_und];

for (int i = 0; i < n_dates; i++)  {
  for (int j = 0; j < n_und; j++) {
    double temp = 0.0;
    k = n_und * i + j;
    for (l = 0; l <= j; l++)
        temp += md_c[n_und * j + l] * md_z[n_und * i + l];
    temp = exp(temp * md_vols[k] + md_drift[k]);
    trajectory[k] = trajectory[k - n_und] * temp;
  }
}
```

- Largely uses destructive updates, reusing same array.
- Deltas: traversed from the back ("antidependence").
- Prefix Sum: appears as flow-dependency (constant distance)

- Difficult (for compiler) to recognize parallelism opportunities.

Performance Comparison & Discussion

	Europ.Call 10^7 samples
Haskell, first version	21.20 sec
Using lists everywhere	16.02 sec
Using arrays for Sobol direction vectors	11.88 sec
Using arrays and recurrence	5.98 sec
C++ version	1.09 sec

(Intel Core i7 2.30GHz)

The jump between 21.2 to 16 sec was due to hoisting constant data-structures at global level.

Sobol Quasi-Random Numbers

Sobol random number generator: generate a sequence of values $\{x^1, x^2, ..., X^N\}$ with low discrepancy $O(logN)$ over interval $[0, 1)$.

- Chose $P \in \mathbb{Z}_2[X]$ primitive polynomial of degree d:
  $P = X^d + a_1 X^{d-1} + ... + a_{d-1} X + 1$,
- Compute the direction vectors:
  $m_i = 2a_1 m_{i-1} \oplus .. \oplus 2^{d-1} a_{d-1} m_{i-d+1} \oplus 2^d m_{i-d} \oplus m_{i-d}$
- To generate the n'th random number, $n = ...b_3 b_2 b_1$,
  $x^n = b_1 v_1 \oplus b_2 v_2 \oplus ...$, where $v_i = m_i / 2^i \in [0, 1)$
- Using Gray code: $...g_3 g_2 g_1 = ...b_3 b_2 b_1 \oplus ...b_4 b_3 b_2$
  $x^{n+1} = x^n \oplus v_c$, where g_c is the rightmost 0 in n's Gray-code rep.
  . . . recursion, can be used for optimisations
- Generalization to S dimensions under $O(log^S N)$ discrepancy.

Recursive Sobol for Optimization

Independent Sobol Formula *vs.* Recurrent Sobol Formula

```
sobolInd  :: Pricing_Data ->
             Index -> [ Elem ]
sobolInd Pricing_Data{..} i =
  let biton = testBit (grayCode i)
      inds  = filter biton
                  [0..SBC-1]
      xorVs vs = foldl' xor 0
                  [vs!j | j<-inds]
  in map xorVs sobol_dirVs


map (sobolInd conf) [1..n]
```

```
sobolRec :: Pricing_Data ->
   [Elem] -> Index -> [Elem]
sobolRec Pricing_Data{..} prev i =
  let bit    = pos_least_sig_0 i
      dirVs = [ vs!bit |
                   vs<-sobol_dirVs ]
  in  zipWith xor prev dirVs

sobolRecMap conf (l,u) =
  let a = sobolInd conf l
  in  scanl (sobolRec conf) a [l..u-1]
sobolRecMap conf (1,n)
```

- Independent: more computation, *but* embarrassingly parallel alg.
- Recurrent: less computation, *but* $log(N)$-depth parallel alg.

- Can we have the benefits of both?

Algorithm-Level Optimizations: Sobol Generator

Optimize Parallelism Via Serial Chunking

```
-- Incorrect for arbitrary l
sobolRecMap conf l =
  let a = sobolInd conf
                   (head l)
  in scanl (sobolRec conf) a
       (map (+ (-1)) (tail l))
```

```
mapChunkList ::([b]->[a])->[b]->Int->[a]
mapChunkList fun l c =
  let sliced :: [[b]]
      sliced = chunkIt c l
      out    = map fun sliced
  in foldl (++) [] (map fun sliced)
```

- Can write mapChunkList to chunk-map a function on an arbitrary list,
- however sobolRecMap is correct only when `l` contains consecutively increasing integers (hence incorrect for arbitrary `l`).

Optimization

Code Ensuring the Invariant

```
sobolRecMap conf (l,u) =
  let a = sobolInd conf l
  in  scanl
        (sobolRec conf)
        a      [l..u-1]
```

Embarrassingly parallel on N/TILE procs: the expensive sobolInd is amortized against TILE (fast) sobolRec computations.
Speedup: 1.8x to 3.75x.

User Expresses Algorithmic Invariants; Compiler Optimizes.

```
-- INVAR: sobolInd c n+1 == sobolRec c (sobolInd c n) n
sobolRec :: Pricing_Data -> [Elem] -> Index -> [Elem]
map (sobolInd conf) [1..n]

-- Compiler Explores The Rich Optimization Space
sobolGen conf n = case (cost_model conf) of
    1 -> doallChunk (sobolRecMap conf) n tile
    2 -> map (sobolInd conf) [1..n]
    3 -> sobolRecMap conf (1,n)
```