Financial Modeling on Parallel Computers using High-Level Programming Languages

Matthew Dixon¹, Jorg Lotze² and Mohammad Zubair³

¹School of Management
University of San Francisco
Email: mfdixon@usfca.edu

²Xcelerit
2832 Upper Pembroke Street
Dublin 2, Ireland
Email: jorg.lotze@xcelerit.com

³Department of Computer Science
Old Dominion University
Norfolk, VA

Thalesians Seminar
Overview

- The need for design space exploration in financial modeling and the consequential emergence of the "implementation gap"
- The vision of how a financial domain specific parallel software framework can address the implementation gap
- Case study: finding performance critical financial applications: detailed insights from using Python, R and C++ for mid to high frequency stochastic volatility modeling
- Towards a parallel implementation of Quantlib for the Intel Xeon Phi coprocessor


Shift to Parallelism

Figure 1.1: Scaling of the processor clock speeds.

Figure: Courtesy of UC Berkeley EECS Parlab
Variety of Parallel Hardware

Multi-Core CPU

CPU Core 0
L1/L2 Cache
Shared L3 Cache
Memory Controller
DRAM

CPU Core 1
L1/L2 Cache

CPU Core 2
L1/L2 Cache

CPU Core 3
L1/L2 Cache

CPU-GPU

CPU Core 0
L1/L2 Cache
Shared L3 Cache
Memory Controller
CPU DRAM
DMA

CPU Core 1
L1/L2 Cache

GPU DRAM

Thread Execution Control Unit

LS

LS

LS

LS

LS

CPU-GPU Block Diagram

Compute Cluster

Interconnection Network

Node 0

Node 1

Node 2

Node 3

Compute Cluster Block Diagram

Figure: Courtesy of UC Berkeley EECS Parlab
Implementation Gap

Figure: Courtesy of UC Berkeley EECS Parlab
Implementation Gap

Enables application design space exploration

Prohibits application design space exploration
Example: design space exploration in financial modeling

There is typically an overwhelming choice of models for a financial instrument, risk factor or derived mathematical variable:

- **Diffusion**: Bachelier, Black-Scholes, CEV, Displaced Diffusion, Hull-White
- **Stochastic Volatility**: Heston, SABR, Displaced Diffusion, Heston, Heston-Hull-White
- **Jump-Diffusion**: Merton, Bates, Bates-Hull-White
- **Levy**: Variance Gamma, Normal Inverse Gaussian
- **Levy+Stochastic Volatility**: Gamma Ornstein-Uhlenbeck and CIR clock

*Problem*: When fitting the model, its parameters can be sensitive to the choice of numerical solver, choice of error measure, and the set of observables $$\Rightarrow$$ *design space exploration is compute intensive*
Research Literature

- K. Keutzer and T. G. Mattson, Introduction to Design Patterns for Parallel Computing, first article, Chapter 8
The Vision of the Research

Vision

A new software framework shall bridge the implementation gap and enable application quantitative financial developers to productively utilize parallel hardware and develop efficient, scalable, portable applications. This software framework shall

- Evolve from a pattern-oriented design for defining scope and vocabulary
- Target critical financial application patterns which are routine computational bottle-necks in financial modeling infrastructure
- Enable financial developers using C++, Python or R to take advantage of various parallel architectures
Our Pattern Language (OPL)

Our Pattern Language

Applications

Structural Patterns
- Pipe-and-Filter
- Agent-and-Repository
- Process-Control
- Event-Based/Implicit-Invocation
- Puppeteer
  - Model-View-Controller
  - Iterative-Refinement
  - Map-Reduce
  - Layered-Systems
  - Arbitrary-Static-Task-Graph

Computational Patterns
- Graph-Algorithms
- Dynamic-Programming
- Dense-Linear-Algebra
- Sparse-Linear-Algebra
- Unstructured-Grids
- Structured-Grids
  - Graphical-Models
  - Finite-State-Machines
  - Backtrack-Branch-and-Bound
  - N-Body-Methods
  - Circuits
  - Spectral-Methods
  - Monte-Carlo

Parallel Algorithm Strategy Patterns
- Task-Parallelism
- Divide and Conquer
- Data-Parallelism
- Pipeline
- Discrete-Event
- Geometric-Decomposition
- Speculation

Implementation Strategy Patterns
- SPMD
  - Kernel-Par.
  - Program structure
- Fork/Join
- Actors
- Vector-Par
- Loop-Parallelism
- Workpile
- Shared-Queue
- Shared-Map
- Partitioned graph
- Distributed-Array
- Shared-Data
- Algorithms and Data structure

Execution Target Architectures
- Distributed Memory MIMD
- Shared Address Space MIMD
- GPU SIMD
- Vector SIMD
Motivation for a Pattern Oriented Design

- Defines the *scope* of the software framework
- Provides a *common vocabulary* for application developers and efficiency programmers
- Enables *modularity* and *comprehensive coverage* of the application domain
- Provides a *language* for describing the software architecture of applications
Example of Patterns in Financial Applications

**Application Patterns**
- Calibration
- Characteristic function
- Grid Based PDE method
- Binomial Trees
- PCA
- Vector Auto-Regression
- GARCH (bekk)
- Factor Modeling

**Computational Patterns**
- Dynamic programming
  - Binomial Trees
  - Calibration
- Sparse linear algebra
  - Grid Based PDE method
  - Factor Modeling
- Spectral methods
  - Characteristic function
- Dense linear algebra
  - PCA
  - GARCH (bekk)
  - Vector Auto-Regression

**Structural Patterns**
- MapReduce
- Iterative Refinement
- Pipe and Filter
Tangible outcome of the research

- Provide a library of financial application patterns which are composed from the computational and structural patterns.

- A *single* optimized implementation of each computational and structural pattern on each architecture (such as GPUs, Intel Xeon Phi, FPGAs) *can be used by multiple application patterns*.

- The library can be made available to C++, Python and R users and the application developer needn’t concern themselves with the details of the parallel implementation.

- Re-use legacy code without having to migrate prototype modeling code into low level programming environments.
Evidence of the Leverage Effect

Daily changes of squared volatility indices versus daily returns. Using the volatility indices as the proxy of volatility, the leverage effect can clearly be seen. Left: S&P 500 data from January 2004 to December 2007, in which the VIX is used as a proxy of the volatility; Right: Dow Jones Industrial Average data from January 2005 to March 2007 in which the Chicago Board Options Exchange (CBOE) DJIA Volatility Index (VXD) is used as the volatility measure.

Stochastic Volatility Modeling

1. Choose a SV model, e.g. Bates Model

2. Estimate the European Option Price

3. Imply the volatility surface from the fitted model prices

4. Trading desks use the implied volatility surface to price non-quoted options and price and hedge exotics
Stochastic Volatility Modeling

1. Choose a SV model, e.g. Bates Model

Definition (Bates Model)

\[
\begin{align*}
\frac{dS_t}{S_t} &= \mu dt + \sqrt{V_t} dW_1^t + (Y - 1)S_t dN_t, \\
\frac{dV_t}{V_t} &= \kappa(\theta - V_t)dt + \sigma \sqrt{V_t} dW_2^t,
\end{align*}
\]

- \( V_t \) is given by a mean reverting square root process constrained by \( 2\kappa \theta - \sigma^2 > 0 \).
- \( N_t \) is a standard Poisson process with intensity \( \lambda > 0 \) and \( Y \) is the log-normal jump size distribution, where \( \mu_j = \ln(1 + a) - \frac{\sigma_j^2}{2}, a > -1 \) and \( \sigma_j \geq 0 \).

2. Estimate the European Option Price

Definition (Bates European Option Pricing Model)

- Call price of a vanilla European option is

\[
C(S_0, K, \tau; \mathbf{p}) = S_0 P_1 - K \exp\{- (r - q)\tau\} P_2,
\]

- \( P_1 \) and \( P_2 \) can be expressed as:

\[
P_j = \frac{1}{2} + \frac{1}{\pi} \int_{0}^{\infty} \text{Re} \left[ \frac{\exp\{-iu\ln K\} \phi_j(S_0, \tau; u; \mathbf{p})}{iu} \right] du, j = 1, 2.
\]

- where the parameter set \( \mathbf{p} := [\theta, \sigma, \kappa, \rho, \nu_0, \mu_j, \sigma_j, \lambda] \).

3. Imply the volatility surface from the fitted model prices

4. Trading desks use the implied volatility surface to price non-quoted options and price and hedge exotics
There is a need to refrequently re-calibrate stochastic volatility models.

**Figure:** The maximum point-wise error across the volatility surface for options on ZNGA against time. The red-line shows the error resulting from calibration of the Bates model every 30 seconds versus calibrating at the start of the period (black-line).
**Definition (Bates Model)**

\[
\frac{dS_t}{S_t} = \mu dt + \sqrt{V_t} dW_t^1 + (Y - 1)S_t dN_t, \tag{1}
\]

\[
\frac{dV_t}{V_t} = \kappa(\theta - V_t)dt + \sigma \sqrt{V_t} dW_t^2, \tag{2}
\]

- $V_t$ is given by a mean reverting square root process constrained by $2\kappa\theta - \sigma^2 > 0$.
- $N_t$ is a standard Poisson process with intensity $\lambda > 0$ and
- $Y$ is the log-normal jump size distribution, where
  \[
  \mu_j = \ln(1 + a) - \frac{\sigma_j^2}{2}, \quad a > -1 \text{ and } \sigma_j \geq 0.
  \]
Definition (European Option Pricing Model)

- Call price of a vanilla European option is

\[
C(S_0, K, \tau; p) = S_0 P_1 - K \exp\{-(r - q)\tau\} P_2,
\]

- \( P_1 \) and \( P_2 \) can be expressed as:

\[
P_j = \frac{1}{2} + \frac{1}{\pi} \int_0^\infty \text{Re} \left[ \frac{\exp\{-iulnK\} \phi_j(S_0, \tau, u; p)}{iu} \right] du, j = 1, 2.
\]

- where the parameter set \( p := [\theta, \sigma, \kappa, \rho, v_0, \mu_j, \sigma_j, \lambda] \).
Definition (Fourier-Cosine Call Price Approximation\textsuperscript{a})


\[
C(S_0, K, \tau; p) \approx K e^{-r\tau} \cdot \text{Re}\left\{ \sum_{k=0}^{N-1} \phi \left( \frac{k\pi}{b-a}; p \right) e^{i k\pi \frac{x-a}{b-a}} U_k \right\},
\]

- \(x := \ln(S_0/K)\) is the log of moneyness
- \(\phi(w; p)\) denotes the Bates characteristic function of the log-asset price,
- \(U_k\) are the payoff series coefficients and
- \(N\) denotes the number of terms in the cosine series expansion (typically 128 will suffice).
Error convergence of FFT versus Fourier-Cosine

Figure: Comparison of the error convergence rates of the Fourier-Cosine (Fang & Oosterlee, 2008), fixed second order Gauss-Legendre quadrature and Carr-Madan FFT methods applied to the Heston pricing model.
Calibration in Python or R

- **Global optimization:** Python and R provide versions of a DE algorithm for global optimization, e.g. *inspyred* and *DEOptim* package \(^1\).

- **Local optimization:** The *scipy.optimize* and *NLopt* module contains various local optimizers in Python and R respectively.

- **Performance:** However, can not quickly calibrate the model which hinders modeling, testing and productionization. *Having to re-hash code into C++ is problematic if the code is still volatile.*

---

The remainder of this talk will describe our experiences with parallel implementations of stochastic volatility model calibration and some preliminary insight into the benefits of a design pattern based approach.
Example 1

Python multi-core CPU Implementation

Hide the parallel implementation details from the user:

1. **Shared-memory**: use the `multi-processing` package to create a pool of $np$ processes on a single multi-core CPU. Assign $n/np$ option model computations to each process.

2. **Distributed-memory**: uses the `mpi4py` packages to launch $mp$ MPI processes across the cluster of multi-core CPUs. A chunk of size $n/mp$ option model computations is assigned to each MPI process.
For calibrating the option price model we consider a sample chain of $n$ option data where the $i^{th}$ chain data has the following properties:

- $S[i]$: Underlying asset price
- $K[i]$: Strike price
- $T[i]$: Maturity
- $\hat{V}[i]$: Market price
Example 1

**Sequential ErrorFunction(p)**

1. \( rmse \leftarrow 0 \)
2. \( \textbf{for} \quad i = 0 \text{ to } n - 1 \quad \textbf{do} \)
3. \( V \leftarrow \text{Price}(S[i], K[i], T[i], p) \)
4. \( \text{diff} \leftarrow \hat{V}[i] - V \)
5. \( rmse \leftarrow rmse + \text{diff} \times \text{diff} \)
6. \( \textbf{end for} \)
7. \( rmse \leftarrow \sqrt{rmse/n} \)
8. \( \textbf{return} \quad rmse \)
Example 1

Parallel Algorithm 1: Parallel-ErrorFunction(p)

```plaintext
1: \( rmse \leftarrow 0 \)
2: Initialize: pool \( \leftarrow \) Pool(processes=\( np \))
3: \# create input parameters for all PRICE computations
4: \( inp \leftarrow [(S[i], K[i], T[i], p), 0 \leq i < n] \)
5: \# assign processes to computations
6: \( res \leftarrow [pool.apply_async(PRICE, ip), ip \in inp] \)
7: \( i \leftarrow 0 \)
8: for \( r \) in \( res \) do
9: \# get results from a process
10: \( V \leftarrow r.get() \)
11: \( diff \leftarrow \hat{V}[i] - V \)
12: \( rmse \leftarrow rmse + diff \times diff \)
13: \( i \leftarrow i + 1 \)
14: end for
15: \( rmse \leftarrow \sqrt{rmse/n} \)
16: 
17: return \( rmse \)
```
Example 1

Parallel Algorithm 2: MPI-ErrorFunction(p)

```plaintext
1: \(\text{rmsel} \leftarrow 0\)
2: Initialize: \(i \leftarrow \text{GetMyRank}()\), \(\text{chunk} \leftarrow n/mp\)
3: \(\text{rmsel} \leftarrow 0\)
4: \textbf{for} \(k = 0\) to \(\text{chunk} - 1\) \textbf{do}
5: \(\text{ko} \leftarrow \text{chunk} \times i + k\)
6: \(V \leftarrow \text{PRICE}(S[ko], K[ko], T[ko], p)\)
7: \(\text{diff} \leftarrow \hat{V}[ko] - V\)
8: \(\text{rmsel} \leftarrow \text{rmsel} + \text{diff} \times \text{diff}\)
9: \textbf{end for}
10: \(\text{rmse} \leftarrow \text{AllreduceMPI}(\text{rmsel})\)
11: \(\text{rmse} \leftarrow \text{SQRT}(\text{rmse}/n)\)
12: 
13: \textbf{return} \(\text{rmse}\)
```
### Performance benchmarks of the parallel algorithms

<table>
<thead>
<tr>
<th>size</th>
<th>SEQ</th>
<th>PAR</th>
<th>MPI</th>
<th>HYB</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>180.43</td>
<td>57.45(3.14)</td>
<td>4.38(41.19)</td>
<td>5.68(31.75)</td>
</tr>
<tr>
<td>128</td>
<td>338.39</td>
<td>103.10(3.28)</td>
<td>6.77(49.97)</td>
<td>6.76(50.04)</td>
</tr>
<tr>
<td>256</td>
<td>706.51</td>
<td>200.63(3.52)</td>
<td>9.25(76.41)</td>
<td>13.08(54.02)</td>
</tr>
<tr>
<td>512</td>
<td>1404.10</td>
<td>383.03(3.67)</td>
<td>11.47(122.43)</td>
<td>21.04(66.74)</td>
</tr>
<tr>
<td>1024</td>
<td>2703.05</td>
<td>639.67(4.23)</td>
<td>19.42(139.19)</td>
<td>33.72(80.16)</td>
</tr>
</tbody>
</table>

**Table:** This table compares the elapsed wall-clock time of the error function in milliseconds between three parallel algorithms. Speedups relative to the sequential version are shown in parentheses.

---

2 Based on a cluster of 32 dual socket Dell PowerEdge R410 nodes with Intel Xeon E5504 processors with 4 cores on each processor
## Example 1

### Overall calibration time (in seconds)

<table>
<thead>
<tr>
<th>size</th>
<th>solver</th>
<th>SEQ</th>
<th>PAR</th>
<th>MPI</th>
<th>HYBRID</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>SLSQP</td>
<td>50.69</td>
<td>16.59</td>
<td>6.05</td>
<td>1.71</td>
</tr>
<tr>
<td>64</td>
<td>LBFGSB</td>
<td>37.74</td>
<td>12.03</td>
<td>0.96</td>
<td>1.27</td>
</tr>
<tr>
<td>64</td>
<td>TNC</td>
<td>126.28</td>
<td>42.16</td>
<td>0.34</td>
<td>2.38</td>
</tr>
<tr>
<td>128</td>
<td>SLSQP</td>
<td>121.72</td>
<td>37.82</td>
<td>3.04</td>
<td>2.57</td>
</tr>
<tr>
<td>128</td>
<td>LBFGSB</td>
<td>81.82</td>
<td>26.00</td>
<td>0.65</td>
<td>1.78</td>
</tr>
<tr>
<td>128</td>
<td>TNC</td>
<td>134.50</td>
<td>43.27</td>
<td>0.68</td>
<td>5.63</td>
</tr>
<tr>
<td>256</td>
<td>SLSQP</td>
<td>535.02</td>
<td>166.93</td>
<td>9.06</td>
<td>9.40</td>
</tr>
<tr>
<td>256</td>
<td>LBFGSB</td>
<td>160.20</td>
<td>50.40</td>
<td>1.83</td>
<td>3.22</td>
</tr>
<tr>
<td>256</td>
<td>TNC</td>
<td>773.95</td>
<td>243.65</td>
<td>5.88</td>
<td>13.37</td>
</tr>
<tr>
<td>512</td>
<td>SLSQP</td>
<td>650.51</td>
<td>192.08</td>
<td>32.24</td>
<td>15.97</td>
</tr>
<tr>
<td>512</td>
<td>LBFGSB</td>
<td>334.76</td>
<td>96.00</td>
<td>13.27</td>
<td>9.36</td>
</tr>
<tr>
<td>512</td>
<td>TNC</td>
<td>1694.05</td>
<td>511.97</td>
<td>45.55</td>
<td>36.61</td>
</tr>
<tr>
<td>1024</td>
<td>SLSQP</td>
<td>1662.01</td>
<td>389.83</td>
<td>48.39</td>
<td>35.63</td>
</tr>
<tr>
<td>1024</td>
<td>LBFGSB</td>
<td>625.59</td>
<td>147.53</td>
<td>17.03</td>
<td>14.15</td>
</tr>
<tr>
<td>1024</td>
<td>TNC</td>
<td>2861.37</td>
<td>684.78</td>
<td>69.97</td>
<td>84.90</td>
</tr>
</tbody>
</table>
Imperative implementation approach

```python
def Error_Function(p0):
    RMSEL = np.float64(0.0)
    n = len(m.chain)
    V = np.zeros(n)
    V_hat = np.array(np.zeros(n), dtype=np.float64)
    inputParameters = [o.getParams(p0) for o in m.chain]
    results = [m.pool.apply_async(HestonCOS, ip) for ip in inputParameters]
    j = 0
    for rt in results:
        V[j] = rt.get()
        V_hat[j] = m.chain[j].getMidPrice()
        j+=1
    r = np.float64(V_hat - V)
    for i in range(n):
        RMSEL = RMSEL + r[i]**2
    RMSE = np.sqrt(RMSEL/n)
    return RMSE
```

- The error function is only optimized for multi-core CPUs
- The type of stochastic volatility model is hard-coded in the error function
Example 1

Computational Pattern: Dynamic Programming

Structural Pattern: Iterative

Structural Pattern: Map-Reduce

Computational Pattern: Spectral Method

Application Pattern: Characteristic Function

Structural Pattern: Map-Reduce

Stochastic Volatility Model

Example: Map-Reduce<Error>

Example: Non-linear solver
Example 1

Example implementation of objective function

Identify Patterns in a Typical Objective Function

A typical objective function called repeatedly by the optimization function

Current guess for the parameter set

An optimization function
SciPy.optimize

The overall computation time is dominated by the time spent in the execution of the objective function

```
def Error_Function(p0):
    for j in range(nkr):
        for i in range(nkc):
            HP_COMPUTED[j][i] = HestonCOS()

    RMSE = sqrt((HP_OBSERVED - HP_COMPUTED)**2)/(nkr*nkc)
    return RMSE
```

```
op = fmin(Error_Function, p0)
```

Introduction  Vision  Case Study  Implementation  Next Steps  Additional Slides
Structural Patterns in an Objective Function

A number of objective functions that appear in financial computation such as calibration, and maximum likelihood problems have a similar structure of map and reduce.

```python
def Error_Function(p0):
    for j in range(nkr):
        for i in range(nkc):
            HP_COMPUTED[j][i] = HestonCOS()

    RMSE = sqrt((HP_OBSERVED - HP_COMPUTED)**2)/(nkr*nkc)
    return RMSE
```

Map(): call some computation for a list of data points. These computations are independent and can be done in parallel.

Reduce(): The output of Map() is reduced using a reduction operator.

Example 1

Python multi-core CPU Implementation

- Hide the parallel implementation details from the user:
  1. **Shared-memory**: use the `multi-processing` package to create a pool of `np` processes on a single multi-core CPU. Assign $n/np$ option model computations to each process.
  2. **Distributed-memory**: uses the `mpi4py` packages to launch $mp$ MPI processes across the cluster of multi-core CPUs. A chunk of size $n/mp$ option model computations is assigned to each MPI process.

- In the following slides:
  - `error_function.py` implements the objective function for the Heston model and is separated from the parallel architecture dependent implementation of map-reduce
  - `mcMapReduce.py` and `clMapReduce.py` implement map-reduce for the shared and distributed memory parallel platforms respectively
def heston(*x):
    n = len(x)
    return pow(HestonCOS(*x[:n-2]) - x[-1], 2.0)

def rmse(x, n):
    return sqrt(x/n)

if __name__ == '__main__':
    fileName = sys.argv[1]
    chain = readIntraDayData(fileName)
    data = [o.getParams() for o in chain]

    mcMapReduce.init(4)
    res = mcMapReduce.eval(heston, data, rmse)
    print(res)
Example 1

error_function.py

```python
1 def heston(*x):
   n = len(x)
   return pow(HestonCOS(*x[:n-2]) - x[-1], 2.0)

5 def rmse(x,n):
   return sqrt(x/n)

9 if __name__ == '__main__':
   fileName = sys.argv[1]
   chain = readIntraDayData(fileName)
   data = [o.getParams() for o in chain]
   clMapReduce.init()
   res = clMapReduce.eval(heston, data, rmse)
   print(res)
```
Example 1

mcMapReduce.py

class mcMapReduce(object):
    @staticmethod
    def init(num_processes=4):
        mcMapReduce.pool = Pool(processes=num_processes)
    @staticmethod
    def __map(f, data):
        results = [mcMapReduce.pool.apply_async(f, row).get() for row in data]
        return results
    @staticmethod
    def __reduce(results, g):
        err = [rt.get() for rt in results]
        return g(sum(err), mcMapReduce.n)
    @staticmethod
    def eval(f, data, g):
        mcMapReduce.n = len(data)
        results = mcMapReduce.__map(f, data)
        return mcMapReduce.__reduce(results, g)
Example 1

cMapReduce.py

class cMapReduce(object):
    @staticmethod
    def init():
        cMapReduce.comm = MPI.COMM_WORLD
        cMapReduce.size = cMapReduce.comm.Get_size()
        cMapReduce.rank = cMapReduce.comm.Get_rank()

    @staticmethod
    def __map(f, data):
        partialSum = np.float64(0.0)
        cMapReduce.chunk = cMapReduce.n / cMapReduce.size
        k0 = cMapReduce.chunk * cMapReduce.rank
        for k in range(0, cMapReduce.chunk):
            partialSum += f(data[k0+k])
        return partialSum

    @staticmethod
    def __reduce(partialSum, g):
        globalSum = np.array(np.zeros(1), dtype=np.float64)
        cMapReduce.comm.Allreduce(partialSum, globalSum, op=MPI.SUM)
        return g(globalSum[0], cMapReduce.n)

    @staticmethod
    def eval(f, data, g):
        cMapReduce.n = len(data)
        results = cMapReduce.__map(f, data)
        return cMapReduce.__reduce(results, g)
Experience with GPUs

- GPUs provide many more cores than can be exploited by the data parallelism in this problem.
- Need to find an additional source of parallelism in order to take full advantage of the parallel architecture.
- Use the OPL based approach to guide a GPU based implementation.
- Example implementation in R.
### Sequential implementation of the calibration program in R

<table>
<thead>
<tr>
<th></th>
<th>AAPL</th>
<th>AMZN</th>
<th>BP</th>
<th>CSCO</th>
<th>GOOG</th>
<th>MSFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEoptim</td>
<td>293</td>
<td>111</td>
<td>69</td>
<td>62</td>
<td>255</td>
<td>70</td>
</tr>
<tr>
<td>nloptr</td>
<td>148</td>
<td>56</td>
<td>35</td>
<td>32</td>
<td>130</td>
<td>36</td>
</tr>
<tr>
<td>ErrorFunction</td>
<td>1.46</td>
<td>0.55</td>
<td>0.34</td>
<td>0.31</td>
<td>1.28</td>
<td>0.35</td>
</tr>
<tr>
<td>Total ErrorFunction</td>
<td>440</td>
<td>166</td>
<td>103</td>
<td>93</td>
<td>385</td>
<td>105</td>
</tr>
<tr>
<td>Total Time</td>
<td>441</td>
<td>167</td>
<td>104</td>
<td>94</td>
<td>386</td>
<td>106</td>
</tr>
<tr>
<td>% ErrorFunction</td>
<td>99.8%</td>
<td>99.4%</td>
<td>99.1%</td>
<td>99.0%</td>
<td>99.7%</td>
<td>99.1%</td>
</tr>
</tbody>
</table>

**Table:** Performance results for the R code in seconds. Each column represents a different option chain.
Example 2

**Sequential implementation of the calibration program in C**

<table>
<thead>
<tr>
<th></th>
<th>AAPL</th>
<th>AMZN</th>
<th>BP</th>
<th>CSCO</th>
<th>GOOG</th>
<th>MSFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEoptim</td>
<td>88</td>
<td>33</td>
<td>21</td>
<td>18</td>
<td>76</td>
<td>22</td>
</tr>
<tr>
<td>Nlopt</td>
<td>44</td>
<td>17</td>
<td>10</td>
<td>9</td>
<td>38</td>
<td>11</td>
</tr>
<tr>
<td>ErrorFunction</td>
<td>0.44</td>
<td>0.17</td>
<td>0.1</td>
<td>0.1</td>
<td>0.38</td>
<td>0.11</td>
</tr>
<tr>
<td>Total ErrorFunction</td>
<td>131</td>
<td>49</td>
<td>30</td>
<td>27.6</td>
<td>113</td>
<td>32</td>
</tr>
<tr>
<td>Total Time</td>
<td>132</td>
<td>50</td>
<td>31</td>
<td>27</td>
<td>114</td>
<td>33</td>
</tr>
<tr>
<td>% ErrorFunction</td>
<td>99.2%</td>
<td>98.0%</td>
<td>96.8%</td>
<td>99.9%</td>
<td>99.1%</td>
<td>97.0%</td>
</tr>
</tbody>
</table>

**Table:** Performance results for the C code in seconds. Based on Intel Core i5 processor.
Efficient offloading of the error function on the GPU requires designing parallel algorithms which keep the pipelines of the processors busy and minimize data movement between the device and the host.

Typically, the model must be re-written in CUDA which requires specialist programming skills which most quants do not possess.

The implementation is typically architecture specific and, as NVIDIA GPUs evolve, may need to be reimplemented to take advantage of increased cores and/or increased memory.

Even implementing the model in OpenCL, it is difficult to design parallel algorithms for different architectures.
Thread Blocks: Scalable Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate
Memory Model

- CPU and GPU have separate memory spaces
  - Data is moved across PCIe bus
  - Use functions to allocate/set/copy memory on GPU
Example 2

**GPU implementation**

- **Map:**
  - Map the Heston model computation on \( n \) blocks of the GPU where \( n \) is the chain size.
  - Each thread block computes the option price for one data point and exploits the task-parallelism in the Heston characteristic function approximation.
  - The number of threads \( N \) in a block is determined by the number of terms in the cosine series expansion.

- **Reduce:**
  - The aggregation of the terms is performed using shared memory and multiple cores of the streaming processor with a tree-like structure.
Example 2

**GPU Implementation**

Diagram showing the implementation of blocks in GPU with variables `bx` and `tx`.
GPU Implementation

Each option in the chain is mapped to a thread block.

Each intermediate result is stored in shared memory.

Each thread computes a term of the Fourier-Cosine series.

Block_1

- bx=0
- tx=0 \rightarrow tx=255

Shared Memory

Block_2

- bx=1
- tx=0 \rightarrow tx=255

Shared Memory

Block_n

- bx=n-1
- tx=0 \rightarrow tx=255

Shared Memory
Example 2

Parallel-Fourier-Cosine(p)

1: *shared memory* smem[]
2:  tx ← threadIdx.x
3:  bx ← blockIdx.x
4:  bd ← blockDim.x
5:  j ← bd
6:  smem[tx] ← CHARACTERISTICFUNCTION(T[bx], p)
7:  for i = 1 to log₂(bd) do
8:      j ← j/2
9:      if tx < j then
11:  end if
12:  end for
13:  if tx = 0 then
14:      V[bx] ← K[bx] × exp(−r₀ × T[bx] × smem[0])
15:  end if
16:  return V[bx]
Parallel reduction using shared memory

Example 2
Example 2

R Wrapper for C++/CUDA implementation

```r
ErrorFunction<-function(z){
  if (!is.loaded('gpuMapReduce')) {
    dyn.load('gpuMapReduce.so')
  }
  RMSE<-.Call("ErrorFunction", as.numeric(p))
  return (RMSE)
}
```
Sample code for performance benchmarking gpusvcalibration

```r
library("gpusvcalibration")
library("DEoptim")
library("nloptr")
chain <- Load_Chain(fileName)
Copy_Data(chain)
Set_Model('Heston') # {"Heston","Bates","VG",CGMY"}
Set_Block_Size(256)

l <- c(eps,eps,eps,-1.0 + eps, eps)
u <- c(5.0-eps,1.0-eps,1.0-eps,1.0-eps,1.0-eps)
args <- list(NP=100, itermax=25)
DEres <- DEoptim(fn=Error_Function, lower=l, upper=u, control=args)

res <- nloptr(x0=as.numeric(DEres$optim$bestmem), eval_f=Error_Function,
              lb = l, ub = u,
              opts=list("algorithm"="NLOPT_LN_COBYLA", "xtol_rel" = xtol))
Dealloc_Data()
```
Performance benchmarks of the GPU implementation

<table>
<thead>
<tr>
<th></th>
<th>AAPL</th>
<th>AMZN</th>
<th>BP</th>
<th>CSCO</th>
<th>GOOG</th>
<th>MSFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/RGPU</td>
<td>1042</td>
<td>992</td>
<td>971</td>
<td>933</td>
<td>1024</td>
<td>961</td>
</tr>
<tr>
<td>C/RGPU</td>
<td>313</td>
<td>297</td>
<td>288</td>
<td>278</td>
<td>303</td>
<td>297</td>
</tr>
<tr>
<td>CGPU/RGPU</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table:** Relative performance of the *RGPU* code to a serial *R* implementation, a serial *C/C++* implementation and a *CGPU* implementation. GPU results are obtained using a NVIDIA Tesla K20c (Kepler architecture).

---

Based on an Intel Core i5 processor and NVIDIA Tesla K20c (Kepler) with 2496 cores
Accompanying resources

- Alpha version of R package\(^4\)

  ```r
  library(devtools)
  install_github("gpusvcalibration", username = "mfrdixon")
  ```

\(^4\)M.F. Dixon, S. Khan and M. Zubair, gpusvcalibration: A R Package for Fast Stochastic Volatility Model Calibration using GPUs, R/Finance, Chicago, 2014
Example 2

Summary of our experiences using R and Python

- Both R and Python offer native support for multi-core and multi-core CPU implementations, although we found the functionality more limited in *R*.
- The *parallel* programming package in R does not currently provide support for GPUs.
- We were unable to keep the implementation of the SV Model in R and were forced to port the model to *CUDA* although we were still able to take advantage of the R environment for calibrating the model.
The Xcelerit platform is a tool which makes it easy for financial application developers to develop high performance applications in C++.

Xcelerit-enabled applications efficiently make use of multi-core CPUs, GPUs, and combinations of these in a grid from a single high-level code-base.

The source code is free from parallel constructs, and the efficient execution is managed automatically.
An Xcelerit processing graph

**SequenceSource**  Option/Term index sequence

\[ i, j - \text{option and term index} \]

**IFourierCosine**  Fourier-Cosine calculation

\[ \text{Fourier term values for all options} \]

**PartialSumReduce**  Sum Fourier terms

\[ \text{Option values} \]

**Error**  Compute RMSE error

**Figure:** An Xcelerit processing graph for the stochastic volatility model calibration.
**Example 3**

```cpp
class IFourierCosine : public Actor {
public:
  Input<int> optIdx_;       // input for option index
  Input<int> ...
};

double HestonCF(...);
double xi(...);

vector<double> T(numOptions), K(numOptions);
double s0, r0, q0, lambda, meanV, sigma, rho, v0;
vector<double> values(numOptions);
lambda = p[0]; meanV = p[1]; ...
// model parameters setup
for (int optIdx = 0; optIdx < numOptions; ++optIdx) {
  values[optIdx] = 0.0;
  for (int termIdx = 0; termIdx < numTerms; ++termIdx) {
    complex<double> j1(0, 1);
    double sigma2 = sigma*sigma;
    double lambda2 = lambda*lambda;
    double c1 = r0*T[optIdx]+(1-exp(...))*...;
    double c2 = 1.0/(8.0*lambda2*lambda)* ...;
    ...
    double U = 2.0/(b-a)*(xi(termIdx,a,b,0,b)-...);
    complex<double> HCF = HestonCF(j*pi/(b-a), ...);
    values[optIdx] += K[optIdx]*...*(unit*HCF*...*U).real();
  }
}
```

**Figure:** Courtesy of Xcelerit
# Performance Benchmarks

<table>
<thead>
<tr>
<th>#terms</th>
<th>Serial</th>
<th>Xcelerit CPU</th>
<th>CUDA</th>
<th>Xcelerit GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>819.0</td>
<td>31.15 (26.3x)</td>
<td>4.50 (182x)</td>
<td>5.16 (159x)</td>
</tr>
<tr>
<td>1,024</td>
<td>1,642.8</td>
<td>61.71 (26.6x)</td>
<td>8.13 (202x)</td>
<td>7.94 (207x)</td>
</tr>
<tr>
<td>2,048</td>
<td>3,323.2</td>
<td>123.06 (27.0x)</td>
<td>15.24 (218x)</td>
<td>13.50 (246x)</td>
</tr>
<tr>
<td>4,096</td>
<td>7,112.7</td>
<td>248.3 (28.6x)</td>
<td>29.40 (242x)</td>
<td>24.48 (291x)</td>
</tr>
</tbody>
</table>

**Table:** This table compares the elapsed wall-clock time of the error function in milliseconds between the serial, CUDA and Xcelerit CPU and Xcelerit GPU using the APPL chain data. The error function is evaluated using varying number of Fourier-Cosine terms. Speedups relative to the sequential version are shown in parentheses.
## Performance Benchmarks

Table: This table compares the elapsed wall-clock time of the error function in milliseconds between the serial, CUDA and Xcelerit CPU/GPU implementations for six single-name equity option chains for 4,096 Fourier terms. Speedups relative to the sequential version are shown in parentheses.

<table>
<thead>
<tr>
<th>Chain</th>
<th>Serial</th>
<th>Xcelerit CPU</th>
<th>CUDA</th>
<th>Xcelerit GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAPL</td>
<td>7,112.7</td>
<td>248.30 (28.6x)</td>
<td>29.40 (242x)</td>
<td>24.48 (291x)</td>
</tr>
<tr>
<td>AMZN</td>
<td>2,701.1</td>
<td>92.77 (29.1x)</td>
<td>11.31 (239x)</td>
<td>10.32 (262x)</td>
</tr>
<tr>
<td>BP</td>
<td>1,679.9</td>
<td>61.17 (27.5x)</td>
<td>7.30 (230x)</td>
<td>7.07 (238x)</td>
</tr>
<tr>
<td>CSCO</td>
<td>1,485.9</td>
<td>52.57 (28.3x)</td>
<td>6.59 (225x)</td>
<td>6.50 (229x)</td>
</tr>
<tr>
<td>GOOG</td>
<td>6,335.6</td>
<td>214.60 (29.5x)</td>
<td>25.88 (245x)</td>
<td>21.64 (293x)</td>
</tr>
<tr>
<td>MSFT</td>
<td>1,708.3</td>
<td>59.42 (28.8x)</td>
<td>7.41 (231x)</td>
<td>7.14 (239x)</td>
</tr>
</tbody>
</table>
### Performance Benchmarks

<table>
<thead>
<tr>
<th>#terms</th>
<th>Solver</th>
<th>Serial time ( (N) )</th>
<th>Xcl CPU time ( (N) )</th>
<th>Xcl GPU time ( (N) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>SLSQP</td>
<td>733 (448)</td>
<td>24.8 (448)</td>
<td>4.0 (448)</td>
</tr>
<tr>
<td></td>
<td>LBFGSB</td>
<td>903 (454)</td>
<td>32.8 (459)</td>
<td>6.8 (487)</td>
</tr>
<tr>
<td></td>
<td>TNC</td>
<td>4,109 (810)</td>
<td>107.0 (678)</td>
<td>49.0 (1,209)</td>
</tr>
<tr>
<td>1,024</td>
<td>SLSQP</td>
<td>1,515 (454)</td>
<td>49.2 (448)</td>
<td>6.2 (448)</td>
</tr>
<tr>
<td></td>
<td>LBFGSB</td>
<td>2,598 (497)</td>
<td>76.8 (478)</td>
<td>7.6 (453)</td>
</tr>
<tr>
<td></td>
<td>TNC</td>
<td>7,750 (783)</td>
<td>331.3 (860)</td>
<td>28.1 (689)</td>
</tr>
<tr>
<td>2,048</td>
<td>SLSQP</td>
<td>3,093 (444)</td>
<td>94.6 (444)</td>
<td>10.2 (444)</td>
</tr>
<tr>
<td></td>
<td>LBFGSB</td>
<td>4,047 (459)</td>
<td>126.9 (458)</td>
<td>14.1 (461)</td>
</tr>
<tr>
<td></td>
<td>TNC</td>
<td>18,123 (838)</td>
<td>357.9 (459)</td>
<td>38.5 (627)</td>
</tr>
<tr>
<td>4,096</td>
<td>SLSQP</td>
<td>6,189 (441)</td>
<td>183.8 (441)</td>
<td>17.8 (441)</td>
</tr>
<tr>
<td></td>
<td>LBFGSB</td>
<td>7,451 (447)</td>
<td>332.6 (486)</td>
<td>33.7 (490)</td>
</tr>
<tr>
<td></td>
<td>TNC</td>
<td>21,349 (627)</td>
<td>805.3 (663)</td>
<td>261.6 (1,339)</td>
</tr>
</tbody>
</table>

**Table:** This table compares the overall time (in seconds) to calibrate the Heston model across the local solvers and sequential and parallel implementations. The number of iterations \( N \) is shown in parentheses.
Summary

- Seek to develop analytics stacks in programming environments which are flexible, scalable, modular and portable
- Identify the critical application patterns in financial applications
  - For example, what are the critical application patterns using Quantlib?
- Prototype a parallel implementation of Quantlib
  - Start by implementing structural and computational patterns in the C++ version of Quantlib for Intel Xeon and Intel Xeon Phi co-processors.
**Our Pattern Language (OPL)**

## Applications

<table>
<thead>
<tr>
<th>Structural Patterns</th>
<th>Model-View-Controller</th>
<th>Iterative-Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipe-and-Filter</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Agent-and-Repository</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process-Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event-Based/Implicit-Invocation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Puppeteer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computational Patterns</td>
<td>Graph-Algorithms</td>
<td>Finite-State-Machines</td>
</tr>
<tr>
<td></td>
<td>Dynamic-Programming</td>
<td>Backtrack-Branch-and-Bound</td>
</tr>
<tr>
<td></td>
<td>Dense-Linear-Algebra</td>
<td>N-Body-Methods</td>
</tr>
<tr>
<td></td>
<td>Sparse-Linear-Algebra</td>
<td>Circuits</td>
</tr>
<tr>
<td></td>
<td>Unstructured-Grids</td>
<td>Spectral-Methods</td>
</tr>
<tr>
<td></td>
<td>Structured-Grids</td>
<td>Monte-Carlo</td>
</tr>
</tbody>
</table>

## Parallel Algorithm Strategy Patterns

- Task-Parallelism
- Divide and Conquer

## Implementation Strategy Patterns

- SPMD
- Kernel-Par.
- Program structure

## Execution Target Architectures

- Distributed Memory MIMD
- Shared Address Space MIMD

## Parallelism

- Data-Parallelism
- Pipeline

## Discrete-Event

- Geometric-Decomposition
- Speculation

## Algorithms and Data structure

- Shared-Queue
- Shared-Map
- Partitioned graph
- Algorithms and Data structure

## Execution Target Architectures

- GPU SIMD
- Vector SIMD
Example of Patterns in Financial Applications

**Application Patterns**
- Calibration
- Characteristic function
- Grid Based PDE method
- Binomial Trees

**Portfolio Analysis**
- PCA
- Vector Auto-Regression
- GARCH (bekk)
- Factor Modeling

**Computational Patterns**
- Dynamic programming
  - Binomial Trees
  - Calibration
- Sparse linear algebra
  - Grid Based PDE method
  - Factor Modeling
- Spectral methods
  - Characteristic function
- Dense linear algebra
  - PCA
  - GARCH (bekk)
  - Vector Auto-Regression

**Structural Patterns**
- MapReduce
- Iterative Refinement
- Pipe and Filter
Asp – A SEJITS for Python [1]