A Pattern Oriented Approach for Designing Scalable Analytics Applications (Invited Talk)

Matthew Dixon
Department of Analytics
University of San Francisco
mfdixon@usfca.edu

Abstract
The biggest gain in fast processing of big-data will most likely be a result of mapping computation onto clusters of machines while exploiting per-processor parallelism by means of vector instructions and multi-threading. As a result, a new generation of parallel computing infrastructure is needed, driven by the need for application scalability through harnessing growing computing resources in the public cloud, in private data centers and custom clusters, and even on the desktop.

This paper uses Our Pattern Language (OPL) to guide the design of a pattern oriented software framework for analytics applications which enables scalability, flexibility, modularity and portability. Using a compute intensive financial application as a motivating example, we demonstrate how following a pattern oriented design approach leads to parallel code in which the domains of concerns for modeling and mapping the computations to the architecture are cleanly delineated, the code is portable across architectures, and accessible to both an application developer and systems developer. In future work, we seek to demonstrate this software framework by architecting and developing a portable and scalable version of quantlib, a popular open-source quantitative finance library.

Categories and Subject Descriptors G.4 [Mathematical Software]: Parallel and vector implementations; D.2.8 [Software Engineering]: Metrics—complexity measures, performance measures

General Terms Algorithms, Performance

Keywords Pattern Oriented Software Frameworks, Quantitative Finance, Python

1. Introduction
Commoditization of computer hardware has enabled software services to be delivered through the cloud, fostering an economy based on delivery and consumption of everything from image storage, video streaming, social networking to personal health monitoring. But the semiconductor industry is banking its future on parallel microprocessors due to the inability to deliver steadily increasing processor frequency gains without pushing power dissipation to unsustainable levels.

The biggest gain in fast processing of big-data will most likely be a result of mapping computation onto clusters of machines while exploiting per-processor parallelism by means of vector instructions and multi-threading. As a result, a new generation of parallel computing infrastructure is needed, driven by the need for application scalability through harnessing growing computing resources in the public cloud, in private data centers and custom clusters, and even on the desktop.

Domain experts Data scientists and quants will likely drive the evolution of this infrastructure as they seek to use general purpose high level programming languages and statistical modeling environments, such as Python and R, for exploration of machine learning and statistical algorithms applied to big datasets. Yet, while the availability of storage combined with commoditization parallel platforms has grown exponentially, the ability of analytics application developers to take advantage of these resources typically requires parallel software architecture design and programming expertise. This leads to the implementation gap problem, illustrated in Figure 1 and explained below, where by the programming environments preferred by application developers are unable to permit scaling of the application to commoditized parallel platforms without reimplementing the code.

1.1 Parallel Software Architecture and Implementation
While traditional software architecture defines the components that make up an application, the communication among components, and the fundamental computations that occur inside components, it does not prescribe how the software architecture is mapped onto the hardware of a parallel computer. With an increasing variety of commoditized parallel platforms available, application developers with little or no parallel computing expertise seek a simple, systematic and comprehensive approach for guiding software architecture design. In addition to achieving scalability, this approach must not compromise flexibility, portability and maintainability of the software.

While the benefits of parallelizing applications are appealing, in order to utilize parallel hardware, the current practice for big compute workloads is to rewrite the applications in low-level languages such as OpenMP, OpenCL or CUDA resulting in long, cumbersome and difficult-to-maintain application code. Furthermore, source code developed for a single desktop GPU platform is not portable to a large cluster. The model of parallel programming associated with the cluster environment is radically different than that of the GPU; In order to map their applications to clusters of machines, programmers need to compose the low-level implementations with distributed programming frameworks such as Hadoop.
and MapReduce. Other factors such as data locality, load balancing and file-system setup also pose significant challenges to developer and researcher productivity and limit code portability thereby impeding experimentation and time to market of applications.

1.2 Summary of Requirements

The culmination of several conversations with analytics practitioners and researchers, partially under the auspices of the workshop on high performance computational finance at SC identifies the following key concerns for application developers:

- Easy migration between single-node experimentation and clusters of machines;
- Maintainability of a single code-base for all architectures;
- Requirement for little or no knowledge of parallel architecture design and parallel programming;
- Enabling rapid prototyping and model design space exploration in a high level general purpose or statistical programming environment; and
- Provision of substantial and mature implementation support infrastructure such as application domain specific libraries.

1.3 Overview

The remainder of this paper is set out as follows. In the following section, we introduce the reader to pattern oriented software frameworks and a conceptual design language for designing such frameworks. In Section 3, we illustrate the design of a pattern oriented software framework for financial analytics. Using a compute intensive financial application as a motivating example, Section 4 illustrates how applying a pattern oriented design approach leads to portability, flexibility, maintainability and performance of the implementation. Section 5 concludes and briefly outlines the next step for building a financial library based on a pattern oriented software framework.

2. Pattern Oriented Software Frameworks

This paper illustrates the use of Our Pattern Language (OPL) [7] to guide the architecture and implementation of financial analytics applications. This process includes considering the structural patterns (also known as architectural styles) that define the overall organization of an application, the basic computational patterns for each stage of the problem, and well as the low level details of the parallel algorithm. OPL is a unified pattern language for designing and implementing parallel software and has formed the basis for pattern oriented frameworks described in [1, 5, 6] which provide programmers with a software environment allowing for both productivity in application development and performance gains of several orders of magnitude using automatic parallelization. Using the selective specialization approach and its separation of concerns, ASPIRE researchers [5] demonstrate how to provide application portability across a variety of parallel platform targets without application code change.

Application frameworks provide a portable environment for application development while still allowing flexibility [5, 6]. One of the main challenges of application driven research relevant to parallel computing is the question of how to consider application characteristics without overfitting to particular specifics of the application. The approach illustrated in this paper identifies the set of core applications, computational and structural patterns in typical financial analytics applications. Once those are defined, we can effectively draw on existing parallel architecture design expertise to efficiently map the software architecture to the hardware. The approach can be likened to a set of recipes that the application developer can use to simplify the parallel architecture design and implementation process.

Gonina et al. [5, 6] explain how pattern oriented software application frameworks bridge the implementation gap and enable analytics application developers to productively utilize parallel hardware and develop efficient, scalable, and portable applications. The primary characteristics of a pattern oriented software framework are that it:

- Evolves from a pattern oriented design for defining scope and vocabulary;
- Targets critical analytics application patterns which have routine computational bottlenecks; and
- Enables application developers using general purpose high level languages to take advantage of various parallel architectures.

Simply put, pattern oriented application frameworks provide the application developer with both the productivity of a high-level language and the performance and scalability of modern parallel hardware. Pattern oriented frameworks are conceptually evolved using OPL, as illustrated in Figure 2.

Pattern Oriented Design

The primary purpose of OPL is to distill the software architecture design process into simple tried and tested recipes, which if followed likely lead to efficient mapping of the application of the architecture. It achieves this by

- Defining the scope of the software framework;
- Providing common vocabulary for application developers and efficiency programmers;
- Enabling modularity and comprehensive coverage of the application domain; and
- Providing a language for describing the software architecture of applications.

3. Pattern Oriented Software Frameworks for Financial Analytics

This section illustrates the conceptual design of a pattern oriented software framework for financial analytics. We first demonstrate how to produce an application domain specific software framework, the scope of which is limited to financial applications in the

Figure 1. A key aspect of the software framework is to bridge the implementation gap between high level languages and efficiency languages. Figure courtesy of UC Berkeley ASPIRE Lab.
capital markets. We emphasize that the purpose of this illustration is not to produce an exhaustive list of application and computational patterns, but rather convey the simplicity and power of the approach of identifying patterns as color-coded building blocks and their assembly. We emphasize that the approach taken in this paper is not limited to financial analytics applications and could just as easily be applied to other computational and data intensive application areas.

The second part demonstrates the implications of good parallel architecture design choices, following OPL to implement applications which are portable across various parallel architectures, modular, flexible and maintainable.

Before continuing further, it is worth noting in passing why a parallel programming model is ideally suited as an example application of the pattern oriented software framework. Large financial instruments require the ability to perform large-scale, compute intensive analytics on-demand. Furthermore, there is a demand for scalable solutions to large-scale testing and comprehensive model exploration in high level programming environments such as Python or R. Examples include backtesting algorithms against historical data and aggregation of results across large datasets, and statistical inference of parameters for parametric distributional representation of statistical distributions. A frequent complaint by a domain expert in financial analytics is the absence of a programming environment that allows programmers to go from experimentation on a single node to a cluster of processors, from processing a sample subset to the entire dataset of content.

Figure 3 illustrates the conceptual design of a pattern oriented software framework for financial applications that provides fairly broad coverage. The figure contains three distinct components:

- **Application patterns**: these are the common mathematical problems underlying many financial models;
- **Computational patterns**: these are the core computational kernels in the model solution technique that are typically the compute bottle-necks; and
- **Structural patterns**: these are the patterns that describe the parallel programming model and, broadly, how the computation and data is mapped to the architecture.

It’s important to note that this approach is somewhat of a unique prospective on financial applications. For one, application developers and domain experts will typically focus on a single application pattern, with the computational patterns and structural patterns typically being addressed by systems developers when performance and scalability issues arise. Thus, pushing the three patterns to the forefront of the application design decreases the likelihood of an implementation gap arising.

In the following section, we shall outline the implications for parallel programming design by following a pattern oriented approach to implementing a financial application.

### 4. Implementation of an Example Financial Application on a Cluster of Multi-Core CPUs

We choose a compute intensive application and first explore the implementation approach in Python for shared memory and distributed memory parallelism. In particular we consider the following two packages:

1. **Shared-memory**: use the `multiprocessing` 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**: use the `mpi4py` packages to launch `mp` MPI processes across the cluster of multi-core CPUs. A chunk of size `n/np` option model computations is assigned to each MPI process.

The application that we choose requires calibrating a chain (or array) of option contracts to financial data. See [2–4] for further details of the application. We introduce some terminology here for ease of exposition. For calibrating the option price model we consider a sample chain of `n` options 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

The calibration algorithm starts with an initial guess of five parameters and iteratively improves the guess using an optimization algorithm until it meets the convergence criteria. A typical organization of this computation involves calling an optimization routine with a pointer to the `ErrorFunction(p)` which estimates the error between market observed option prices $\hat{V}[i]$ and prices calculated
using the model, \( V[i] \), for the current guess of the parameter set \( p \). More specifically, the \( \text{ErrorFunction}(p) \) computes option prices using the option model for a list of tuples \( \langle K[i],\, T[i] \rangle \), \( 0 \leq i < n \) using the current estimates of the five parameters \( p \) and compares it with the corresponding data in \( \hat{V}[i] \). In our discussion, we focus on the parallel implementation of the \( \text{ErrorFunction}(p) \) as it dominates the overall computation.

A high level description of the sequential version of the \( \text{ErrorFunction}(p) \) is given in Algorithm 1. Note that for reasons of keeping the description simple, we have avoided some subtleties of the implementation. The bottle-neck is in the option pricing function \( \text{Price} \).

### Algorithm 1 Sequential-ErrorFunction(p)

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

#### 4.1 Parallel Implementation

For the parallel implementation, we explored two approaches using Python with \textit{multiprocessing} and \textit{mpi4py} packages. These are described by Algorithms 2-3. In the first approach, we just use the \textit{multiprocessing} package to parallelize the option model computation \( (V, \text{line 3, Algorithm 1}) \). For this we start \( np \) processes and assign a process to compute \( n/np \) of option price computations (see Algorithm 2). Essentially this parallelizes the loop of Algorithm 1 among \( np \) processes. The value of \( np \) is determined by the number of processors or cores available on the computer. Once a value of an option price is computed by a process it needs to be aggregated with values generated by other processes for computing the root mean square error (rmse). This approach is very similar to OpenMP based parallelization and is suitable for shared memory architectures like a multi-core system.

The second approach just uses the \textit{mpi4py} package and is suitable for a cluster of computers. In the MPI based approach, defined by Algorithm 3, we start \( np \) processes on a node and assign several option model computations to a process. More specifically, we assign a \( \text{chunk} = n/np \) of option model computations to a MPI process. A local \( \text{rmse} \) value is generated by each process, see line 8 of Algorithm 3. The computation for \( \text{rmse} \) requires aggregation of local \( \text{rmse} \) computed for each of the processes and is implemented using MPI collective computation primitives (line 10 of Algorithm 3).

#### 4.2 Portability

In this section we illustrate the usage of OPL to identify the structural and computational patterns and separate the domains of concerns between the application and systems developer. The essential idea is the following - the domain expert should be able to easily change the definition of the model without concern for the underlying architecture. In the following implementations, we show how by simply modifying two lines of code in the main function, the application developer is able to deploy the same code on a shared memory architecture or distributed memory architecture. Note that the specification of the model, with a spectral transformation as the computational pattern, is kept separate from the structural pattern, i.e. map and reduce. An additional benefit is that no low level programming is required by the application developer, although a systems developer would likely need to write the code for mapping the computation to the architecture.

\textit{error_function.py} implements the objective function for the stochastic volatility (Heston) model and, crucially, is separated from the parallel architecture dependent implementation of mapreduce \textit{mcMapReduce.py} and \textit{clMapReduce.py} implement mapreduce for the shared and distributed memory parallel platforms respectively.

```python
def hestonCOS(x):
    n = len(x)
    return pow(HestonCOS(x; \mu, \sigma, \lambda, \kappa), n)

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

if __name__ == '_main_':
    fileName = sys.argv[1]
    chain = readIntraData(fileName)
    data = [g.getParams() for g in chain]
    cMapReduce.init()
    res = cMapReduce.eval(heston, data, rmse)
    print(res)
```

Figure 4. Implementation of the model, the reduction operator and the main function.

#### 4.3 Performance Results

We evaluated the performance for our implementation of the \textit{ErrorFunction()} on a cluster of 32 dual socket Dell PowerEdge R410 nodes. The Dell PowerEdge R410 has two Intel Xeon E5504 processors with 4 cores on each processor for a total of 8 cores on a node. The sequential and shared memory parallel implementations
were evaluated on one of the Intel Xeon E5504 processors of the cluster. We evaluated the MPI approach on all the 32 nodes of the cluster.

**Data** An end-of-day snapshot of S&P 500 European options on August the 8th yields over 3000 mid-prices for use as a test set. To avoid calibrating to illiquid instruments, we exclude in-the-money calls and puts and further remove quotes with zero bids. This reduces the calibration set to approximately 1700 mid-prices. For convenience of benchmarking, we randomly select subsets of this chain in powers of 2 up to 1024 contracts. The short rate and dividend yield are set to 0.02% and 1.96% respectively. 128 terms are used in the Fourier-Cosine method for all benchmarking.

**Earlier experiments** In earlier experiments, detailed in [2], we observed: (a) the overall time to calibrate the stochastic volatility model is dominated by the ErrorFunction() routine both in sequential and parallel cases (more than 95%), and (b) the local solver is sensitive to the ordering of data in the RMSE calculation resulting in variation in number of iterations needed to converge for sequential and MPI based versions. For these reasons, we report average execution time of a single iteration of ErrorFunction() in Table 1.

**Results** Table 1 compares the timing results in milliseconds of the model error function across the three approaches along with the speedups against the serial version shown in the parentheses. We observe that the timing of the sequential version scales linearly and that a chain of 1024 options costs 2.7s per error function evaluation. Recall that Algorithm 2 is run by eight processes in a multiprocessor pool on a single node and Algorithm 3 is run by eight MPI processes on each node. Overall for the range of problem sizes tested, we observe that the parallel efficiency of Algorithm 2 is the highest of the parallel algorithms and that this increases with chain size. The drastic increase in parallel efficiency of Algorithm 3 with chain size shows that Algorithm 3 is best suited to large chain sizes.

### Table 1. This table compares the elapsed wall-clock time of the error function in milliseconds between two parallel algorithms. Speedups relative to the sequential version are shown in parentheses.

<table>
<thead>
<tr>
<th>size</th>
<th>SEQ</th>
<th>PAR</th>
<th>MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>180.43(57.43(1.14))</td>
<td>4.38(41.19)</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>338.39(103.10(3.28))</td>
<td>6.77(49.97)</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>706.51(200.63(5.32))</td>
<td>9.25(76.41)</td>
<td></td>
</tr>
<tr>
<td>512</td>
<td>1404.10(833.03(3.67))</td>
<td>11.47(122.43)</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>2703.05(639.67(4.23))</td>
<td>19.42(139.19)</td>
<td></td>
</tr>
</tbody>
</table>

### 5. Conclusion

This paper uses Our Pattern Language (OPL) to guide the design of a pattern oriented software framework for building financial applications which enables scalability, flexibility, modularity and portability. Using a compute intensive financial application as a motivating example, we demonstrate how following a pattern oriented approach leads to parallel code in which the domains of concerns for modeling and mapping the computations to the architecture are cleanly delineated, the code is portable across various architectures, and accessible to both an application developer and systems developer. In future work, we seek to demonstrate this software framework by architecting and developing a portable and scalable version of quantlib, a popular open-source quantitative finance library.

### Acknowledgments

The authors would like to gratefully acknowledge the support of Hanweck Associates and the International Securities Exchange in providing access to a data sample from the ISE/Hanweck Premium Hosted Database.

### References