Parallel Computation of Value at Risk using the Delta-Gamma Monte Carlo Approach

Nan Zhang, Ka Lok Man and Eng Gee Lim

Abstract—**Value at Risk (VaR) is a widely used measure for qualifying risk. It specifies the maximum loss a portfolio can suffer in N days' time with confidence level X%. A bank's portfolio usually consists of thousands of risky assets. To be able to quickly evaluate the VaR of a large portfolio is challenging. In this work, we have developed a multi-threaded program on shared-memory multi-processor systems that calculates the VaR of large portfolios. We follow the delta-gamma Monte Carlo approach. To replace the matrix-matrix multiplications in the gamma term by matrix-vector multiplications we diagonalised the gamma term and represented it by the productions of its eigenvectors and eigenvalues. We used Intel's MKL functions for the operations on the matrices and vectors. Joe and Kuo's pre-computed direction numbers were fed into MKL's random number generator to generate Sobol' quasi-random sequence. The simulation stage was parallelised by creating POSIX threads and assigning each created thread an equal number of the trial paths. The program was tested by a portfolio of 4000 assets. The parallel simulation ran more than 4 times as fast as the simulation parallelised by MKL's vectorised automatic multi-threading functions.**

Index Terms—**Value at Risk, Monte Carlo simulation, Delta-gamma approximation, Parallel computing, Sobol' sequence**

I. Introduction

VALUE at risk is a single, summary statistical measure of possible portfolio losses that would result from an adverse movement in the risky assets constituting the portfolio. Since its first use by J.P. Morgan it has become a benchmark for measuring the risk exposures of financial portfolios. It has become widely used by corporate treasurers, fund managers and financial regulators. It is used by the Basel Committee in setting capital requirements for banks throughout the world [1].

Given a time horizon (N days) and a confidence level (X%) the VaR (Y dollars) of a portfolio is the maximum value in dollars that the portfolio can loss in the next N days with the confidence level of X%. Put it in another way, there is a probability of X% that the loss of the portfolio will not exceed Y dollars in the next N days. Bank regulators require banks to calculate VaR for market risk with N = 10 and X = 99 [2].

There are a number of approaches to calculate the VaR of a portfolio consisting of non-linear financial products. See, for example, [3], [4], [5] for discussions about and comparisons between these methods. In our work, we take the delta-gamma Monte Carlo method to calculate the VaR of non-linear portfolios. This approach is a compromise between the parametric delta-gamma method and the full Monte Carlo method. It is more accurate than the parametric delta-gamma method, and, yet, computationally more efficient than the full Monte Carlo method. In a general delta-gamma VaR approach, the delta term reflects the part of the portfolio loss/gain that linearly depends on the value change in the underlying assets. The quadratic gamma term approximates the part of the portfolio loss/gain that non-linearly depends on the value change in the underlying assets.

Usually, a bank's portfolio consists of thousands of linear and non-linear risky assets. To be able to evaluate its VaR in a short time is important to decision markings. To this end, we have developed a general parallel program that is able to calculate the VaR of portfolios consisting of thousands of risky assets. The program is targeting at x86 shared-memory multi-processor systems. With POSIX threads the simulation stage of the computation is explicitly parallelised. The implementation was tested by a portfolio of 4000 risky assets on a computer powered by an Intel Xeon E5-1660 – a processor having 6 cores, supporting 12 threads and running at 3.3GHz. For comparison purposes we also created a program in which the simulation stage is parallelised by Intel MKL functions' default multi-threading ability. The timing results demonstrated that in completing the simulation stage of the computation our parallel program was more than 4 times as fast as the program using MKL's automatic multi-threading.

Organisation of the rest of the paper: Section II presents the overall delta-gamma Monte Carlo approach in calculating the VaR of a portfolio and the diagonalisation optimisation. Section III discusses how to generate Sobol' quasi-random sequence using Intel MKL's routine when the dimension of the randoms exceeds 40. Section IV describes how the simulation stage of the computation is parallelised by POSIX threads. Section V presents the test results and the comparisons between our parallel program and the program using MKL's automatic multi-threading. Section VI draws the conclusion and points out a direction for potential future work.

II. The delta-gamma Monte Carlo VaR approach

We assume the time t positions in a portfolio of N risky assets are $X_0, X_1, \ldots, X_{N-1}$, respectively. Over a very short time period Δt, such as 1 day's time, we assume the returns $\boldsymbol{R} = (\Delta X_0/X_0, \Delta X_1/X_1, \ldots, \Delta X_{N-1}/X_{N-1})$ of the assets follow a multivariate normal distribution with zero mean vector and the variance-covariance matrix Σ_S, and, so, we

Manuscript received September 10, 2013; revised September 30, 2013. This work was supported by the XJTLU Research Development Fund under Grant 10-03-08.

Nan Zhang is with the Department of Computer Science and Software Engineering, Xi'an Jiaotong-Liverpool University, China. Email: nan.zhang@xjtlu.edu.cn

Ka Lok Man is with the Department of Computer Science and Software Engineering, Xi'an Jiaotong-Liverpool University, China, and Baltic Institute of Advanced Technologies, Vilnius, Lithuania. Email: ka.man@xjtlu.edu.cn

Eng Gee Lim is with the Department of Electrical and Electronic Engineering, Xi'an Jiaotong-Liverpool University, China. Email: enggee.lim@xjtlu.edu.cn

have $\boldsymbol{R} \sim \mathcal{N}^N(\boldsymbol{0}, \Sigma_S)$. We use $P(t) = \sum_{i=0}^{N-1} X_i$ to denote the time t value of the portfolio and $P(t + \Delta t)$ to denote the updated value of the portfolio at the end of the Δt time period. The value change ΔP of the portfolio over this time period is what we are interested in. If $\delta_i = \partial P(t)/\Delta X_i$, $i = 0, 1, \ldots, N-1$, is the sensitivity of the portfolio with respect to the change in value X_i, and $\Gamma_{ij} = \partial^2 P(t)/\partial X_i \partial X_j$, $i,\ j = 0, 1, \ldots, N-1$ is the sensitivity of the change in δ_i with respect to the change in X_j, the value change $\Delta P = P(t+\Delta t) - P(t)$ can be approximated by the Taylor polynomials as

$$\Delta P = \sum_{i=0}^{N-1} \delta_i \Delta X_i + \frac{1}{2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \Gamma_{ij} \Delta X_i \Delta X_j. \tag{1}$$

Using the return adjusted deltas and gammas $\tilde{\delta}_i = \delta_i X_i$ and $\tilde{\Gamma}_{ij} = \Gamma_{ij} X_i X_j$, Equation 1 can be written in terms of the returns $R_i = \Delta X_i / X_i$ as

$$\Delta P = \sum_{i=0}^{N-1} \tilde{\delta}_i R_i + \frac{1}{2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \tilde{\Gamma}_{ij} R_i R_j. \tag{2}$$

In matrix form Equation 2 is

$$\Delta P = \tilde{\boldsymbol{\delta}}^{\mathrm{T}} \boldsymbol{R} + \frac{1}{2} \boldsymbol{R}^{\mathrm{T}} \tilde{\Gamma} \boldsymbol{R}, \tag{3}$$

where $\tilde{\boldsymbol{\delta}} = (\tilde{\delta}_0, \tilde{\delta}_1, \ldots, \tilde{\delta}_{N-1})$ and $\tilde{\Gamma} = [\tilde{\Gamma}_{ij}]$. The variance-covariance matrix Σ_S and the adjusted gamma matrix $\tilde{\Gamma}$ are symmetric.

Using the Cholesky factorisation we can find a lower triangular matrix C and C's transpose C^{T} such that the variance-covariance matrix $\Sigma_S = CC^{\mathrm{T}}$. The correlated returns $\boldsymbol{R} = (R_0, R_1, \ldots, R_{N-1})$ can then be expressed in terms of the independent standard normal variables $\boldsymbol{Z} = (Z_0, Z_1, \ldots, Z_{N-1}) \sim \mathcal{N}^N(\boldsymbol{0}, \boldsymbol{1})$ as $\boldsymbol{R} = C\boldsymbol{Z}$. So, in terms of $\boldsymbol{Z}$ Equation 3 becomes

$$\Delta P = \tilde{\boldsymbol{\delta}} C \boldsymbol{Z} + \frac{1}{2} \boldsymbol{Z}^{\mathrm{T}} (C^{\mathrm{T}} \tilde{\Gamma} C) \boldsymbol{Z}. \tag{4}$$

To reduce the matrix-matrix multiplications in the gamma term to matrix-vector multiplications, we follow the diagonalisation approach presented in [6] and [7]. Using spectral decomposition we can find matrices U and Λ such that $1/2 C^{\mathrm{T}} \tilde{\Gamma} C = U \Lambda U^{\mathrm{T}}$, where

$$\Lambda = \begin{vmatrix} \lambda_0 & & & \\ & \lambda_1 & & \\ & & \ddots & \\ & & & \lambda_{N-1} \end{vmatrix}$$

is a diagonal matrix and U is an orthogonal matrix ($UU^{\mathrm{T}} = I$) whose columns are eigenvectors of $1/2 C^{\mathrm{T}} \tilde{\Gamma} C$. The diagonal elements λ_i of Λ are eigenvalues of $1/2 C^{\mathrm{T}} \tilde{\Gamma} C$. We substitute $U \Lambda U^{\mathrm{T}}$ into the gamma term in Equation 4, and it becomes

$$\begin{aligned} \frac{1}{2} \boldsymbol{Z}^{\mathrm{T}} (C^{\mathrm{T}} \tilde{\Gamma} C) \boldsymbol{Z} &= Z^{\mathrm{T}} U \Lambda U^{\mathrm{T}} Z \\ &= (U^{\mathrm{T}} Z)^{\mathrm{T}} \Lambda (U^{\mathrm{T}} Z) \\ &= \boldsymbol{H}^{\mathrm{T}} \Lambda \boldsymbol{H}, \end{aligned} \tag{5}$$

Algorithm 1: Computing delta-gamma VaR in Monte Carlo simulation.

```
Input: Variance-covariance matrix $\Sigma_S$, vector $\boldsymbol{X}$ of $N$ asset positions,
       vector $\boldsymbol{\delta}$, matrix $\Gamma$, percentile $\alpha$, number $M$ of experiments.
Output: Portfolio value change vector $\boldsymbol{V}$, 1 day $(1-\alpha)$ value at risk
        VaR.
1  begin
2      $\tilde{\delta}_i \leftarrow \delta_i X_i$, $\tilde{\Gamma}_{ij} \leftarrow \Gamma_{ij} X_i X_j$, // $i, j = 0, 1, \ldots, N-1$
3      Find lower triangular matrix $C$ such that $\Sigma_S = CC^{\mathrm{T}}$
4      Find matrices $\Lambda$ and $U$ such that $1/2 C^{\mathrm{T}} \tilde{\Gamma} C = U \Lambda U^{\mathrm{T}}$
5      $\boldsymbol{B} \leftarrow U^{\mathrm{T}} C^{\mathrm{T}} \tilde{\boldsymbol{\delta}}$
6      for $i = 0$ to $M-1$ do
7          Generate vector $\boldsymbol{Z}$ from Sobol' sequence
8          $\boldsymbol{H} \leftarrow U^{\mathrm{T}} \boldsymbol{Z}$
9          $\Delta P \leftarrow 0$
10         for $j = 0$ to $N-1$ do
11             $\Delta P \leftarrow \Delta P + B_j H_j + \lambda_j H_j^2$
12         $V_i \leftarrow \Delta P$
13     Sort $\boldsymbol{V}$ in ascending order
14     VaR $\leftarrow V_{\lfloor \alpha M \rfloor}$
15 end
```

where $\boldsymbol{H} = U^{\mathrm{T}} Z$. In order to be able to use vector $\boldsymbol{H}$ in computing the delta term as well, we transform the delta term as

$$\begin{aligned} \tilde{\boldsymbol{\delta}} C \boldsymbol{Z} = (C^{\mathrm{T}} \tilde{\boldsymbol{\delta}})^{\mathrm{T}} \boldsymbol{Z} &= (C^{\mathrm{T}} \tilde{\boldsymbol{\delta}})^{\mathrm{T}} U U^{\mathrm{T}} \boldsymbol{Z} \\ &= (C^{\mathrm{T}} \tilde{\boldsymbol{\delta}})^{\mathrm{T}} U \boldsymbol{H} \\ &= (U^{\mathrm{T}} C^{\mathrm{T}} \tilde{\boldsymbol{\delta}})^{\mathrm{T}} \boldsymbol{H} \end{aligned} \tag{6}$$

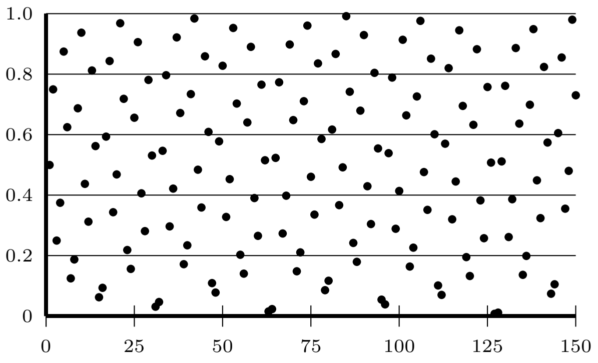
If we let vector $\boldsymbol{B} = U^{\mathrm{T}} C^{\mathrm{T}} \tilde{\boldsymbol{\delta}}$, the delta term can be written as $\boldsymbol{B}^{\mathrm{T}} \boldsymbol{H}$. The value change ΔP of the portfolio, therefore, is

$$\begin{aligned} \Delta P &= \boldsymbol{B}^{\mathrm{T}} \boldsymbol{H} + \boldsymbol{H}^{\mathrm{T}} \Lambda \boldsymbol{H} \\ &= \sum_{i=0}^{N-1} (B_i H_i + \lambda_i H_i^2). \end{aligned} \tag{7}$$

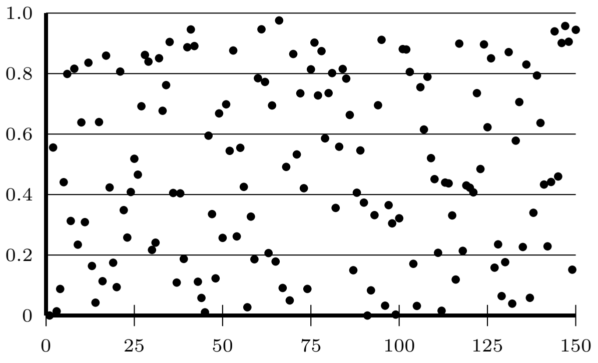
In Monte Carlo simulation we generate M N-dimensional vectors $\boldsymbol{Z} = (Z_0, Z_1, \ldots, Z_{N-1})$. Using each $\boldsymbol{Z}$ we calculate a ΔP. We then sort the M ΔP's in ascending order. For a given percentile α, such as 5%, the 1 day $(1-\alpha)$ VaR is the ΔP at the $\lfloor \alpha M \rfloor$-th position in the sequence. The procedure is summarised in Algorithm 1.

III. Gaussian random generation from Sobol' sequence

As in [8], to generate Gaussian random vectors $\boldsymbol{Z}$, we first generate a sequence of Sobol' quasi-random numbers uniformly distributed over the interval $[0, 1)$. We then convert this sequence to standard normal randoms by the Box-Muller method [9]. The Sobol' quasi-random sequences were first proposed by Sobol' [10], and was subsequently implemented by Bratley and Fox [11]. Quasi-random number generators are often the preferred choice for generating high-dimensional sequences of uniformly distributed variates. Fig. 1 shows two groups of 150 uniformly distributed random variates over the interval $[0, 1)$. One was generated by the Sobol' sequence and the other by the pseudo-random generator MRG32k3a [12]. It can be seen from Fig. 1(a) that the Sobol' numbers are more evenly distributed over the interval.



(a) Sobol'



(b) MRG32k3a

Fig. 1: Comparison between 150 quasi-random numbers and 150 pseudo-random numbers.

TABLE I: MKL functions used in computing the matrices and the vectors before the simulation stage.

MKL function	Operation
LAPACKE_spotrf	Find C such that $\Sigma_\mathrm{S} = CC^\mathrm{T}$
cblas_strmm (twice)	Compute $C^\mathrm{T}\tilde{\Gamma}C$ (C is triangular)
LAPACKE_ssyevd	Find U,Λ for $1/2C^\mathrm{T}\tilde{\Gamma}C = U\Lambda U^\mathrm{T}$
mkl_simatcopy	Compute U^T
cblas_sgemm, cblas_sgemv	Compute $\boldsymbol{B} = U^\mathrm{T}C^\mathrm{T}\tilde{\boldsymbol{\delta}}$

Unlike the authors in [8], in our implementation, we use the Sobol' quasi-random number generator provided by Intel's Math Kernel Library (MKL) [13], which is based on Algorithm 659 of Bratley and Fox [11]. To enable the generator to generate 4000-dimensional random vectors we use the direction numbers pre-computed by Joe and Kuo [14], [15] as parameters passed to the MKL routine.

IV. The parallel computing

In the implementation, before the M-trial simulation starts, we use Intel's MKL functions to compute the matrices and the vectors whose computation is presented in Algorithm 1, line 3 to 5. These MKL functions all support automatic multi-threading and are able to perform the operations in parallel on shared-memory multi-processor machines. These functions are able to dynamically adjust the degree of parallelism according to how much resource is available in the system. The use of these MKL functions is summarised in Table I.

The Monte Carlo simulation is explicitly parallelised. If the number of processors in the system is c, the M-trial simulation is evenly divided into c fractions. Each processor is allocated M/c trials of the simulation to finish. To avoid resource contention between MKL's automatic multi-threading and our explicit multi-threading we kept the MKL's multi-threading switched off during the simulation stage. To avoid threads from migrating between different processors, we bound each thread onto a distinct processor at the time the thread was created. To be able to generate Sobol' quasi-random number in parallel each created thread skipped a certain number of points and started to generate from a certain position in the stream. If we create c threads and bind each of them onto one of the c processors, the i-th thread, $i \in [0, c-1]$, will start to generate from the (iNM/c)-th position of the stream.

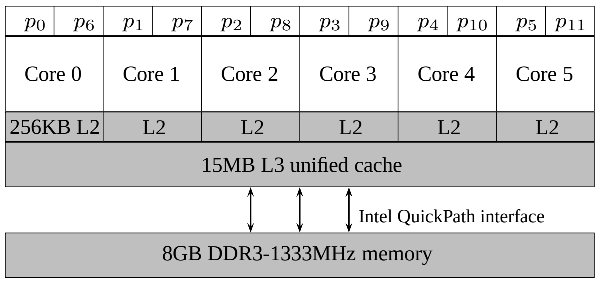

p_0 p_6 p_1 p_7 p_2 p_8 p_3 p_9 p_4 p_{10} p_5 p_{11}
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
256KB L2
L2
15MB L3 unified cache
Intel QuickPath interface
8GB DDR3-1333MHz memory

Fig. 2: Intel Xeon E5-1660.

TABLE II: Runtimes in our explicit and MKL's automatic parallel executions. Times were measured in seconds.

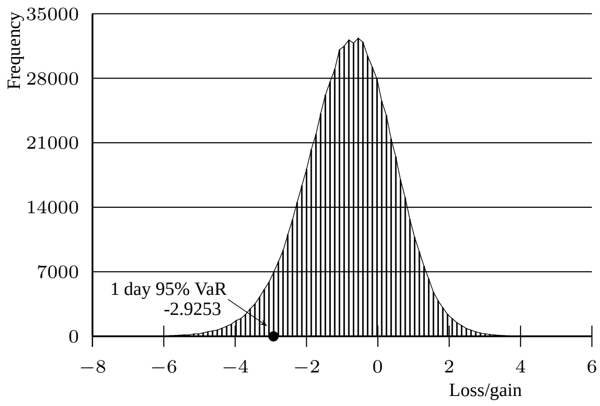
Operation	Explicit	Automatic
Cholesky factorisation $\Sigma_\mathrm{S} = CC^\mathrm{T}$	0.237	0.233
Compute $C^\mathrm{T}\tilde{\Gamma}C$	0.460	0.463
Find U, Λ, U^T for $1/2C^\mathrm{T}\tilde{\Gamma}C = U\Lambda U^\mathrm{T}$	2.051	2.061
Compute $\boldsymbol{B} = U^\mathrm{T}C^\mathrm{T}\tilde{\boldsymbol{\delta}}$	0.429	0.434
M trials	356.9	1472.8
Sort	0.108	0.094

V. Experimental results

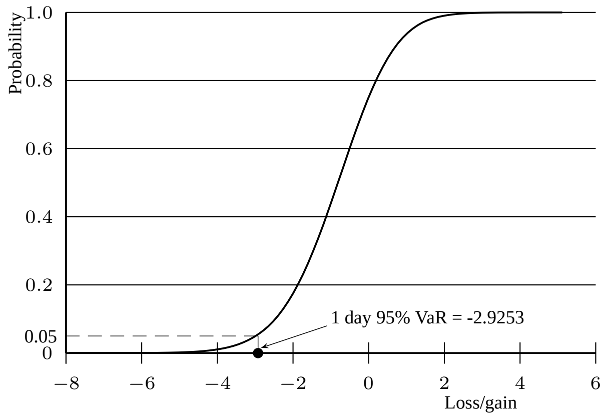
The algorithm presented in Algorithm 1 and its parallel version were implemented in C/C++. The executable binary code was generated by Intel's C/C++ compiler icpc 13.0.1 for Linux. The machine we used in the tests was powered by an Intel Xeon E5-1660 processor (Fig. 2) running on Ubuntu Linux 13.04 64-bit version. The processor runs at 3.3GHz. It has 6 cores supporting 12 threads.

To test the implementations, as in [8], we set $N = 4000$ and $M = 750000$. We set the elements in vector $\boldsymbol{X}$ all to ones. We ran the simulation in our explicit parallel scheme with 12 threads, and compared its runtime with the simulation in 1 thread but parallelised automatically by the multi-threaded MKL functions. The timing results in seconds are reported in Table II. Note that the matrix and vector calculation in the tests were performed by MKL functions with automatic multi-threading. So the two sets of runtimes in those operations were very close. But in the simulation stage the speedup of our explicit parallelisation was 4.13 against the MKL's multi-threading.

From the portfolio value change vector computed by the explicit parallel implementation the 1 day 95% VaR was -2.9253. The largest loss and gain was -7.99 and 5.18, respectively. Between these two extreme values we created 150 equal-sized bins. We counted the number of times the value change fell into each of these bins. The frequency



(a) Density function



(b) Cumulative distribution function

Fig. 3: Density and cumulative distribution functions of the portfolio value change.

graph is plotted in Fig. 3(a), and the cumulative distribution function of the value change in Fig. 3(b).

VI. Conclusion

We have presented our work on computing the value at risk using the delta-gamma Monte Carlo approach. The algorithm consists of two stages – the preparation and the simulation. In the preparation stage matrices and vectors needed by the trials of the simulation are pre-computed. In each single trial of the simulation quasi-random numbers are generated to compute a portfolio value change. After all the trials are completed, all the computed value changes are sorted. A certain value of change is then chosen as the value at risk for a given percentile.

To avoid repetitive matrix-matrix multiplications during the trials, the gamma term matrix is diagonalised in such a way that it is represented by the productions of its eigenvectors and eigenvalues. Such representation enables the matrix-matrix multiplication in the gamma term to be replaced by less expensive matrix-vector multiplications. Because the gamma term is evaluated in every single trial, this diagonalisation saves much execution time.

We use Intel's MKL functions to generate Sobol' quasi-random numbers and to perform all the matrix and vector computations. On shared-memory multi-processor machines these MKL functions are automatically multi-threaded. In our implementation all the matrix and vector computations in the preparation stage are implicitly parallelised by these multi-threaded functions. However, for the simulation we switched off the automatic multi-threading option. Instead, we created POSIX threads to explicitly parallelise the trials of the simulation. We bound each created thread onto a distinct processor, working on an equal fraction of the total number of the trials. Experiential results demonstrated that, compared to MKL's automatic multi-threading, our explicit parallelisation was more than 4 times as fast as.

For future work, we will adapt this delta-gamma Monte Carlo VaR algorithm for general purpose graphics processors to further speedup the simulation.

References

[1] J. C. Hull, *Options, Futures, and Other Derivatives*, 8th ed. Prentice Hall, 2012, ch. 21.
[2] ——, *Risk Management and Financial Institutions*, 1st ed. Pearson Education, 2007, ch. 8.
[3] G. Castellacci and M. J. Siclari, "The practice of Delta-Gamma VaR: Implementing the quadratic portfolio model," *European Journal of Operational Research*, vol. 150, no. 3, pp. 529–545, Nov. 2003.
[4] M. Britten-Jones and S. M. Schaefer, "Non-Linear Value-at-Risk," *European Finance Review*, vol. 2, pp. 161–187, 1999.
[5] I. N. Khindanova and S. T. Rachev, "Value at Risk: Recent Advances," in *Handbook on Analytic-Computational Methods in Applied Mathematics*. CRC Press LLC, 2000.
[6] P. Glasserman, *Monte Carlo Methods in Financial Engineering*. Springer, 2004, ch. 9.
[7] A. Feuerverger and A. C. M. Wong, "Computation of value-at-risk for nonlinear portfolios," *Journal of Risk*, Oct. 2000.
[8] M. F. Dixon, J. Chong, and K. Keutzer, "Accelerating Value-at-Risk estimation on highly parallel architectures," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 8, pp. 895–907, 2012.
[9] G. E. P. Box and M. E. Muller, "A Note on the Generation of Random Normal Deviates," *Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610–611, 1958.
[10] I. M. Sobol', "On the distribution of points in a cube and the approximate evaluation of integrals," *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 86–112, 1967.
[11] P. Bratley and B. L. Fox, "Algorithm 659: Implementing Sobol's quasirandom sequence generator," *ACM Transactions on Mathematical Software*, vol. 14, no. 1, pp. 88–100, Mar. 1988.
[12] P. L'Ecuyer, "Good Parameter Sets for Combined Multiple Recursive Random Number Generators," *Operations Research*, vol. 47, no. 1, pp. 159–164, 1999.
[13] *Intel Math Kernel Library Reference Manual - MKL 11.0 Update 5*, Intel Corporation, 2013, Document Number: 630813-061US. http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/mklman/mklman.pdf.
[14] S. Joe and F. Y. Kuo, "Constructing Sobol Sequences with Better Two-Dimensional Projections," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2635–2654, Jun. 2008.
[15] ——, "Remark on algorithm 659: Implementing Sobol's quasirandom sequence generator," *ACM Transactions on Mathematical Software*, vol. 29, no. 1, pp. 49–57, Mar. 2003.