

# A multithreaded Java implementation of Monte Carlo simulation for Value-at-Risk estimation

Paweł Borowiecki

0009-0008-6825-4038

Maria Curie-Skłodowska University (UMCS)

ul. Pl. Marii Curie-Skłodowskiej 5, 20-031 Lublin, Poland

Beata Bylina

0000-0002-1327-9747

Maria Curie-Skłodowska University (UMCS)

ul. Pl. Marii Curie-Skłodowskiej 5, 20-031 Lublin, Poland

Email: beata.bylina@mail.umcs.pl

**Abstract**—The continuous development of computers has made software that performs calculations in various fields of science, including economics, significantly more efficient.

This paper presents two parallel implementations developed in Java and a performance analysis of Monte Carlo simulations for Value-at-Risk estimation, using geometric Brownian motions and a log-normal distribution of returns. The first implementation variant utilizes data parallelism and parallel loops. The second implementation variant, in addition to data parallelism and parallel loops, also utilizes task parallelism and the fork-join model.

Software tests were conducted on three computers equipped with Intel processors. The main area of research was the dependence of simulation execution time and speedup on the number of threads and data size. The program's performance was accelerated by more than 42 times.

## I. INTRODUCTION

ADVANCES in computer technology in recent years have significantly contributed to increasing the efficiency of computations in various fields of science, including economics. This article presents two variants of a multithreaded implementation of Monte Carlo simulation for Value-at-Risk estimation. Value-at-Risk is a key element in financial analysis, as it is one of the factors that allows financial institutions to estimate market risk [1]. One method for estimating Value-at-Risk is Monte Carlo simulation. Obtaining an accurate result using this method involves lengthy computation time [2], so it is worthwhile to employ parallel programming to reduce this time.

To date, numerous studies have been conducted on how to accelerate the Monte Carlo simulation process for Value-at-Risk estimation using Intel AVX [3] and high-performance GPU accelerators [4], as well as the CUDA [5] and Intel MIC architectures [6]. Parallel programming in C/C++ was used to estimate Value-at-Risk using the Delta-Gamma Monte Carlo method [7].

This article aimed to develop an implementation using parallel programming to accelerate the overall simulation process. Java was chosen as the programming language, as it provides numerous tools for handling multithreading. The first implementation variant utilizes data parallelism and parallel loops. The second implementation variant, in addition to data parallelism and parallel loops, also utilizes task parallelism and the fork-join model. Furthermore, few references in the

literature that combine the topic of Value-at-Risk with programming use the Java language.

The software tests were conducted on three server-class computers equipped with Intel processors, each of which has a different architecture and number of cores.

The remainder of the article is structured as follows. Section II provides an introduction to the economic and mathematical theory, which familiarizes the reader with the definition of Value-at-Risk and the basic mathematical formulas on which the implemented simulation is based. Section III describes the details of both implementations, highlighting mechanisms from the `java.util.concurrent` package. In addition, the functioning strategies of both implementations are presented, highlighting the types of parallelism and parallel programming patterns used. Section IV contains a detailed description of the test environment, as well as the results of the software testing, focusing on how time and speedup depend on the number of threads and the data size.

## II. MONTE CARLO SIMULATION IN VALUE-AT-RISK ESTIMATION

Value-at-Risk is a statistical measure of the risk of a decline in market value, based on the current market value, which depends on a specific time horizon and tolerance level [8]. One method for estimating Value-at-Risk is Monte Carlo simulation, which involves repeatedly simulating a random process for given financial instruments in an effort to account for various situations that may arise in the financial market [8]. This allows for the calculation of return rates from predefined probability distributions. This approach enables the reflection of the distribution of the entire investment portfolio, which allows for the estimation of Value-at-Risk [1].

One of the fundamental pillars of Monte Carlo simulation is the use of historical data — past observed asset prices. Furthermore, this method requires the use of an appropriate stochastic process. A correlation matrix is also used, created by multiplying the covariance matrix of the assets, which has previously undergone Cholesky decomposition, by a vector of generated standard normal variables. The process of calculating Value-at-Risk in this way boils down to the following steps:

- 1) Selection of the stochastic process and parameters.

- 2) Generation of pseudo-random numbers used to calculate individual asset prices.
- 3) Calculation of asset values over a given time horizon.
- 4) Calculating potential losses and sorting them.
- 5) Determining Value-at-Risk as a quantile [8].

In this study, geometric Brownian motions combined with a log-normal distribution of returns was used to simulate price paths, and the future values of the assets in the portfolio were calculated using the following formula:

$$S(t) = S(0) \exp \left( \left[ \mu - \frac{1}{2}\sigma^2 \right] t + \sigma W(t) \right), \quad (1)$$

where:

- $S(0)$  — initial value of the asset,
- $\mu$  — drift,
- $\sigma$  — volatility,
- $W$  — standard Brownian Motion.

The above formula can be generalized to the following form:

$$S(t_{i+1}) = S(t_i) \exp \left( \left[ \mu - \frac{1}{2}\sigma^2 \right] (t_{i+1} - t_i) + \sigma \sqrt{t_{i+1} - t_i} Z_{i+1} \right), \quad i = 0, 1, \dots, n-1, \quad (2)$$

where:

- $S(t_{i+1})$  — asset price at time  $t_{i+1}$ ,
- $Z_{i+1}$  — independent standard normal variable [9].

The marginal distribution  $S(t)$  is an exponential normal distribution, known as a log-normal distribution, which allows returns to be calculated using the formula:

$$R_i = \ln \left( \frac{S(t_j)}{S(t_{j-1})} \right) \quad (3)$$

where:

- $R_i$  — return on the  $i$ -th asset at time  $t_{j-1}$ ,
- $S(t_j)$  — asset price at time  $t_j$ .

Potential losses are calculated by subtracting the expected value of the asset from its initial price. These losses are then sorted, and Value-at-Risk is calculated based on this.

### III. IMPLEMENTATION

Two variants of the parallel implementation of the previously described simulation were created. Both implementations use parallel loops. Additionally, the second variant also uses the fork-join model. The algorithm takes as input an array of asset prices in the investment portfolio, an array of asset weights, a confidence level, the number of days for which the simulation is to be executed, the number of price path simulations for each asset, and the number of threads on which the program is to run.

The primary data type used in both implementations is `BigDecimal`, which provides high decimal precision and control over number rounding — a feature that is extremely important from an economic standpoint, as it allows for highly accurate results.

The class responsible for generating pseudorandom numbers when creating a vector of independent normal variables is the `SplittableRandom`, whose instances are passed to threads by calling the `split` method on the parent object, which is not subsequently used to generate pseudorandom values, allowing threads to safely use the newly created objects [10]. Additionally, the Box-Muller transformation was used to obtain values from the normal distribution.

In both implementations, potential losses are sorted by calling the `parallelSort` method of the `Arrays` class. However, ensuring that the covariance matrix is positive definite and performing the Cholesky decomposition is executed sequentially what does not significantly affect the program's efficiency.

The first implementation primarily focuses on data parallelism. The most important class is `MonteCarloSimulationMultithreaded`, which contains the `runSimulation` method responsible for starting the simulation and calling the other methods tasked with calculating the individual parameters necessary to perform the price path simulation. These methods launch thread classes, most of which implement the `Runnable` interface, with the exception of the class involved in calculating the initial portfolio value, which implements the `Callable` interface. The use of this interface necessitated the use of the `ExecutorService` and `Future` interfaces. Each thread receives its own uniform data range on which it performs specific operations, although the last thread created may receive a larger data range if an even distribution is not possible. This approach prevents a race condition for resource access. The `CountDownLatch` class was used as the synchronization mechanism, which notifies the main thread when it is possible to resume work.

In the second parallel implementation, task parallelism was used in addition to data parallelism. The most important class in this implementation is the `MonteCarloSimulationMultithreaded2` class, which, as in the first implementation, has a `runSimulation` method. A `ForkJoinPool` thread pool is created with the number of threads allocated for the entire simulation. Additionally, the `CompletableFuture` class is used, which allows for the asynchronous execution of the following tasks:

- creating and decomposing the covariance matrix,
- calculating volatility,
- calculating the initial portfolio value.

If only two threads are allocated for the entire simulation, the decomposed matrix is created on one thread, while the operations responsible for calculating volatility and the initial portfolio value are performed on the other thread. The methods responsible for performing the tasks listed above do not create separate threads — the method execution process occurs sequentially.

The rest of the methods in the simulation work identically to the first implementation, with the exception of price path simulation, for which the `parallel` method of the `IntStream`

interface is used, allowing the previously created thread pool to fully manage the assignment of tasks to threads.

#### IV. PERFORMANCE ANALYSIS AND TEST ENVIRONMENT

The tests were conducted on three computers designated C1, C2, and C3, each equipped with a different server-class processor, with each subsequent model has better parameters than the previous one. Their detailed specifications are as follows:

- C1 (2x Intel Xeon Gold 5218R) — 20 cores / 40 threads, Cascade Lake Refresh architecture (14 nm);
- C2 (2x Intel Xeon Gold 6342) — 24 cores / 48 threads, Ice Lake-SP architecture (10 nm);
- C3 (2x Intel Xeon Platinum 8358) — 32 cores / 64 threads, Ice Lake-SP architecture (10 nm).

The testing process began by running the sequential implementation, followed by each of the parallel implementations, executed sequentially on a number of threads ranging from 1 to the maximum number of logical threads available. The simulation duration was the total execution time of the runSimulation method. Time measurement was performed using the `System.nanoTime()` method. The speedup is the ratio of the execution time of the sequential version of the program to the execution time of the parallel version. The program was primarily tested using ten different datasets (asset portfolios and weight matrices), each of which had a different size. The remaining input parameters, with the exception of the number of threads such as the confidence level were constant for all datasets.

The symbol  $x$  in the legend of the figures presented later in this article denotes the multiple of the data size relative to the smallest tested data size. This means that in the case of a size of  $10x$ , the data size is ten times larger than the smallest tested data size.

For all computers, the execution time of both implementations is inversely proportional to the number of threads used in the simulation, which is fully consistent with Amdahl's law, which states that the maximum theoretical speedup has already been achieved [11].

Figure 1 shows the relationship between acceleration and the number of threads for the first version of the parallel implementation on computer C1. A sharp increase in performance is noticeable up to 20 threads, followed by a certain plateau, likely caused by the fact that the processor had been using physical cores up to that point, and after exceeding 20 threads, it begins to utilize Hyper-Threading, which causes threads to compete for resources at the level of the physical processor core. From around 40 threads, the performance begins to rise slowly again, as the second processor is also utilized, though the increases are no longer as sharp as at the beginning, because the processor must access resources from RAM.

In Figure 2, which shows the performance of the second parallel implementation on computer C1, a sharp increase in performance up to about 20 threads can be observed, followed by a slight plateau at around 40 threads. This may

The relationship between the speedup and the number of threads for the first parallel implementation on computer C1

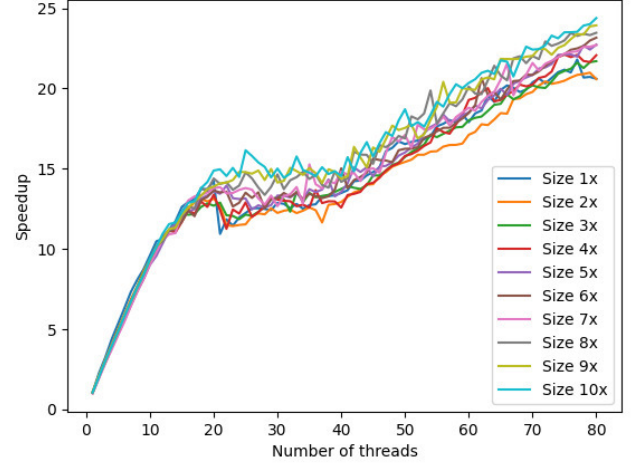


Fig. 1. The relationship between the speedup and the number of threads for the first parallel implementation on computer C1

be caused by competition for thread resources on the same cores, which could be due to work-stealing algorithm in the `ForkJoinPool` thread pool. Next, there is another sharp increase in performance due to the use of a second processor, followed by a plateau at a high number of threads, for the same reasons as in the case of the first plateau.

The relationship between the speedup and the number of threads for the second parallel implementation on computer C1

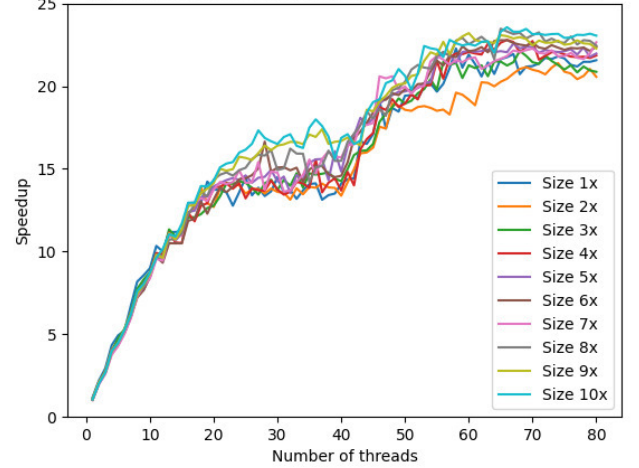


Fig. 2. The relationship between the speedup and the number of threads for the second parallel implementation on computer C1

In Figures 3 and 4, which show the relationship between performance and the number of threads for the first parallel implementation on computers C2 and C3, a rapid increase in performance can be observed until the number of threads equals the number of processor cores, followed by a period of very large fluctuations resulting from the start of more intensive use of the second processor and overheads associated with

the NUMA architecture, such as access to remote memory and increased cache coherence traffic. This is followed by a slight decline in performance and a steady increase that is no longer as rapid, which is likely due to reaching memory bandwidth limits and increased synchronization costs between threads.

processor and the low load on the second processor. However, as the number of threads increases and the second processor's workload grows, further increases in acceleration can be observed. Noteworthy is the drastic drop in performance on the C2 computer for just over 60 threads with the smallest dataset, likely caused by excessive overhead associated with thread management by the thread pool.

The relationship between the speedup and the number of threads for the first parallel implementation on computer C2

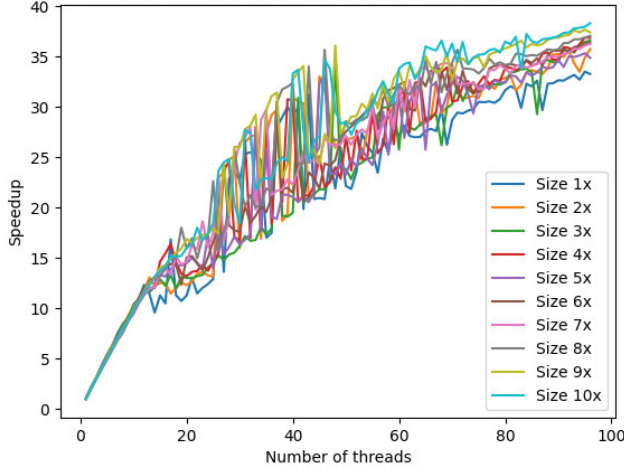


Fig. 3. The relationship between the speedup and the number of threads for the first parallel implementation on computer C2

The relationship between the speedup and the number of threads for the second parallel implementation on computer C2

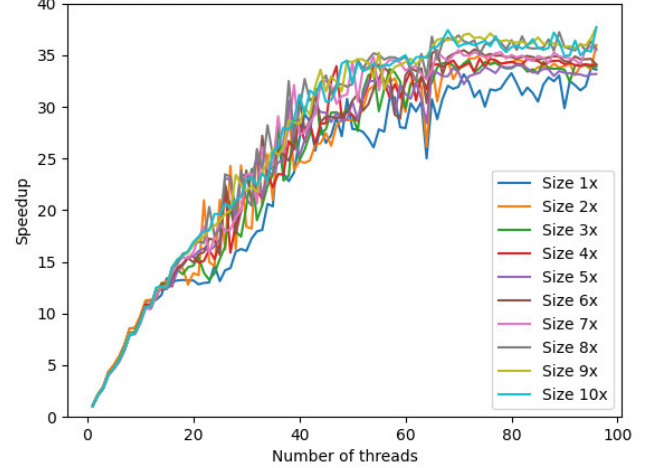


Fig. 5. The relationship between the speedup and the number of threads for the second parallel implementation on computer C2

The relationship between the speedup and the number of threads for the first parallel implementation on computer C3

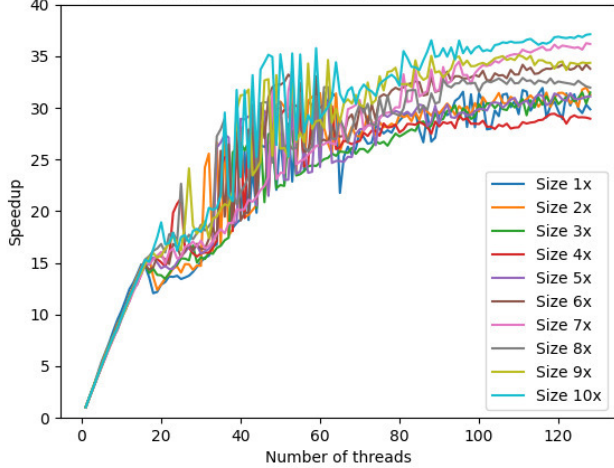


Fig. 4. The relationship between the speedup and the number of threads for the first parallel implementation on computer C3

The relationship between the speedup and the number of threads for the second parallel implementation on computer C3

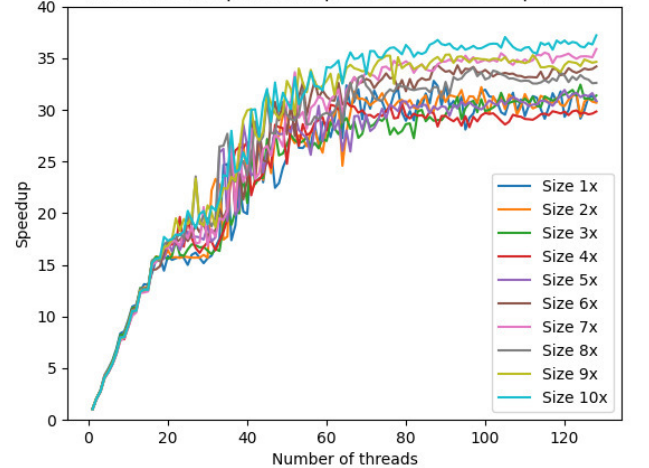


Fig. 6. The relationship between the speedup and the number of threads for the second parallel implementation on computer C3

Figures 5 and 6 show the relationship between performance scaling and the number of threads for the second implementation on computers C2 and C3. A steep increase in performance is observed as long as the number of threads does not exceed the number of physical cores in the processor. Once the number of cores is exceeded, the rate of increase slows down and fluctuations appear, related to the high load on the first

Summarizing the results from all computers, the shortest runtime for the maximum number of threads for data sizes ranging from 1x to 10x was recorded for the first implementation on computer C2, although for small datasets, the results of both implementations on computers C2 and C3 are

very similar, as shown in Figure 7. In contrast, the longest runtime was measured for both implementations running on computer C1.

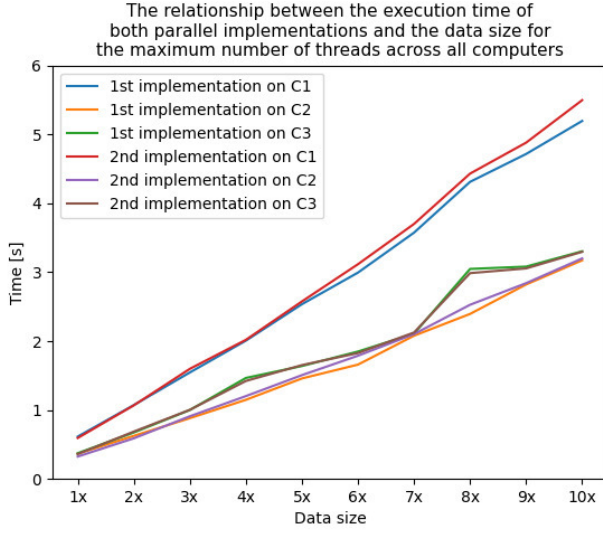


Fig. 7. The relationship between the execution time of both parallel implementations and the data size for the maximum number of threads across all computers

The greatest speedup for data sizes ranging from 1x to 10x was observed for the first version of the implementation on computer C2, with the exception of the smallest dataset, for which the greatest speedup was observed for the second version of the implementation on computer C2, as shown in Figure 8.

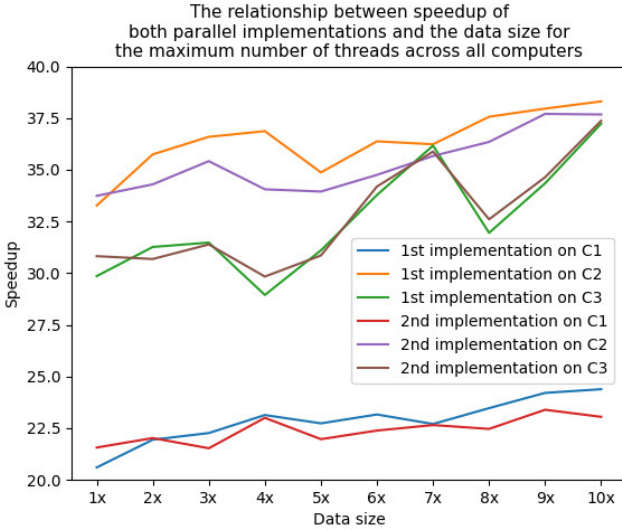


Fig. 8. The relationship between speedup of both parallel implementations and the data size for the maximum number of threads across all computers

The speedups measured for both implementations across data sizes ranging from 1x to 10x, when run on the C3 computer, show a similar trend, and for the 7x data size,

they match the programs running on computer C2. It should be noted that computer C3, which has the largest number of cores, did not achieve the highest speedup for this data range in every case. Therefore, both implementations were additionally tested for data sizes of 15x, 20x, 25x, and 30x. The results are presented in Tables I and II. For a data size of 15x, the highest speedup was recorded for the second implementation variant, running on computer C2. However, for data sizes of 20x, 25x, and 30x, the greatest speedups were recorded for both implementations running on computer C3, and the speedup for the second implementation for 30x data exceeded 42x, indicating that for this computer, the input data size must be sufficiently large.

TABLE I  
SPEEDUP OF THE FIRST IMPLEMENTATION FOR DATA SIZES OF 15x, 20x, 25x AND 30x ON ALL COMPUTERS

| Speedup of the first implementation for data sizes of 15x, 20x, 25x and 30x on all computers |         |         |         |
|----------------------------------------------------------------------------------------------|---------|---------|---------|
|                                                                                              | C1      | C2      | C3      |
| 15x                                                                                          | 19.7235 | 29.6798 | 27.5808 |
| 20x                                                                                          | 20.6108 | 31.0633 | 32.5892 |
| 25x                                                                                          | 25.2650 | 35.7620 | 37.5054 |
| 30x                                                                                          | 25.8888 | 38.0322 | 40.9800 |

TABLE II  
SPEEDUP OF THE SECOND IMPLEMENTATION FOR DATA SIZES OF 15x, 20x, 25x AND 30x ON ALL COMPUTERS

| Speedup of the second implementation for data sizes of 15x, 20x, 25x and 30x on all computers |         |         |         |
|-----------------------------------------------------------------------------------------------|---------|---------|---------|
|                                                                                               | C1      | C2      | C3      |
| 15x                                                                                           | 20.5501 | 32.5164 | 32.3631 |
| 20x                                                                                           | 22.9676 | 34.4352 | 38.1631 |
| 25x                                                                                           | 25.8211 | 37.3377 | 40.5544 |
| 30x                                                                                           | 25.7501 | 38.3150 | 42.7427 |

The smallest speedups were recorded for both implementations running on computer C1. For all data sizes, the first implementation was faster, with the exception of the smallest data size.

## V. CONCLUSION

This paper demonstrated how parallelizing the implementation of Monte Carlo simulations for Value-at-Risk estimation, using geometric Brownian motions and a log-normal return distribution, contributed to faster computation times. The performance of both implementations depended on the types of parallelism and parallelism patterns employed. Tests were conducted on three server-class computers with varying numbers of cores and architectures, focusing on the dependence

of execution time and speedup on the number of threads and the size of the input data.

The tests confirmed the superiority of multithreaded implementations over the sequential version, as the speedup ranged from just over  $19\times$  to more than  $42\times$ .

The greatest performance gains for the  $1x$ – $10x$  datasets were observed on computer C2 for the first implementation variant, with the exception of the smallest dataset, for which the second implementation variant, run on the same computer, was the fastest. In contrast, for larger datasets, the greatest performance gains were recorded for the implementations run on computer C3. The programs running on computer C1 performed the slowest, particularly the second implementation variant, with the exception of the result for the smallest dataset.

Research has confirmed that the effectiveness of parallel implementations depends on the number of processor cores, as well as on how the operating system and the Java virtual machine manage threads. The architecture of the computer system, as well as the type and size of the input data, are also key factors.

Future research on the parallel implementation of Monte Carlo simulations for Value-at-Risk estimation could focus on testing the software not only on various CPUs but also on GPUs. It would be worthwhile to explore other programming languages and compare which implementation yields the greatest performance boost.

## REFERENCES

- [1] Alexander J. McNeil, Rüdiger Frey, Paul Embrechts. *Quantitative Risk Management: Concepts, Techniques and Tools - Revised Edition*. Princeton University Press, 2015.
- [2] John C. Hull. *Options, Futures and other derivatives*. Pearson Education, 2015.
- [3] Liyanage, DNSS and Fernando, GVMPPA and Arachchi, DDMM and Karunathilaka, RDDT and Perera, Amal Shegan. Utilizing Intel advanced vector extensions for Monte Carlo simulation based value at risk computation. Elsevier, 2017. <http://dx.doi.org/10.1016/j.procs.2017.05.156>
- [4] Wen-Mei W. Hwu. *Gpu Computing Gems Jade Edition*. Morgan Kaufmann Publishers, 2011. <http://dx.doi.org/10.1016/C2010-0-68654-8>
- [5] Wei Wu. *Acceleration of Monte Carlo Value at Risk Estimation Using Graphics Processing Unit (GPU)*. City University of New York Academic Works, 2010.
- [6] Tiansheng Wen, Rui Mao, Cheng Tan. Parallel Monte Carlo Simulation of VaR Calculation Based on Intel MIC Architecture. Institute of Electrical and Electronic Engineers, 2020. <http://dx.doi.org/10.1109/ICBDIE50010.2020.00116>
- [7] Nan Zhang, Ka Lok Man, Eng Gee Lim. Parallel Computation of Value at Risk using the Delta-Gamma Monte Carlo Approach. *Proceedings of the International MultiConference of Engineers and Computer Scientists*, 2014.
- [8] Philippe Jorion. *Value At Risk The new Benchmark for Managing Financial Risk*. The McGraw-Hill Companies, 2007.
- [9] Paul Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2003. <http://dx.doi.org/10.1007/978-0-387-21617-1>
- [10] Oracle Corporation. *Java Platform, Standard Edition & Java Development Kit Version 22 API Specification*. Oracle America Inc, 2024.
- [11] Jamie Allen Roland Kuhn, Brian Hanafey. *Reactive Design Patterns*. Manning Publications Co, 2017.