DOI 10.34185/1562-9945-3-158-2025-08
UDC 004.8:62-50

G. Shvachych, P. Shcherbyna, O. Kabachenko, I. Olishevskyi, P. Ishchuk

DEVELOPMENT AND EXPLORATION OF PARALLEL TECHNOLOGIES IN STOCHASTIC PROGRAMMING TASKS

Abstract. This research examines parallel technologies for modeling tasks using the Monte-Carlo method. The actuality of these studies is explained by the fact that the Monte-Carlo method has had and continues to have a significant impact on the development of computational mathematics. It is shown that the main essence of the method lies in the random simulation of a large number of scenarios and statistical processing of the results, which explains the inherent possibility of its parallelization. It is noted that since individual iterations of the Monte-Carlo method are typically independent of one another, they can be easily distributed among several threads or nodes of a cluster system. This makes the method ideal for parallel and distributed computing. The main aim of the research is to highlight peculiarities of parallelizing computations in solving a wide range of applied tasks. Calculation schemes that ensure increased performance and speed are presented. The effectiveness of the proposed approach is illustrated by studies and graphical interpretations of convergence and approximation of the developed approach.

Keywords: parallel computing, stochastic modeling, random process, method convergence, approximation, computation scheme.

Problem statement. The Monte-Carlo method has significantly influenced and continues to influence on the development of computational mathematics methods, particularly numerical integration methods. In solving many problems, it successfully combines with other computational methods, complementing them.

The essence of the Monte-Carlo method is as follows. Let it be required to find the value *a* of a certain studied value. To do this, a random variable X is chosen, whose mathematical expectation is equal to *a*, i.e. $M(X) = a$.

Practically, this is done as follows: *n* experiments are conducted, resulting in *n* possible values of *X*; their arithmetic mean is calculated: $\bar{x} = \frac{\sum x_i}{n}$, and $\bar{x}$ is considered as the estimate (approximate value) a* of the desired number a, that is, $a \simeq a^* = \bar{x}$.

Since the Monte-Carlo method requires large number of experiments, it is often referred to as the method of statistical trials. The theory of this method shows how to choose the most appropriate random variable *X* and find its possible values. In particular, methods for reducing the dispersion of the involved random variables are being developed, which leads to

© Shvachych G., Shcherbyna P., Kabachenko O., Olishevskyi I., Ishchuk P., 2025

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

a decrease in the error that occurs when replacing the sought mathematical expectation *a* with its estimate, *a**.

The most important characteristic of the method is the estimation of its error. Since this is of significant importance, let's consider some approaches to error estimation for the Monte-Carlo method to understand its essence.

Let *n* independent trials (n possible values of X were experimentally implemented) be conducted to obtain the estimate *a** of the mathematical expectation *a* of the random variable *X*, and let the sample mean $\overline{x}$ be found based on these trials and accepted as the desired estimate when $a^* = \overline{x}$. It is clear that in the case of repeating the experiment, different possible values of *X* will be obtained, leading to different mean values and, consequently, a different estimate *a**. Hence, it follows that obtaining an exact estimate of the mathematical expectation is impossible. Naturally, the question arises about the magnitude of the permissible error. For clarity, let's limit ourselves to finding only the upper bound δ of the permissible error with a given probability (reliability) *γ*, namely: $P(\left|\overline{X} - a\right| \leq \delta) = \gamma$.

It is worth noting that the upper bound of the error δ, which we are interested in, is nothing other than the "accuracy of the estimate" of the mathematical expectation based on the sample mean value in confidence intervals. Let's consider the following three cases.

1. The random variable *X* is normally distributed, and its standard deviation σ is known.

In this case, with confidence γ, the upper bound of the error is

$$\delta = \frac{t_\gamma \cdot \sigma}{\sqrt{n}}, \quad (1)$$

where *n* is the number of trials (experimental values of X); t_γ is the argument value of the Laplace function at which $\Phi(t_\gamma) = \gamma$; σ is the known standard deviation of the random variable *X*.

2. The random variable *X* is normally distributed, and its standard deviation σ is unknown.

In this case, the upper bound of the error with reliability γ is determined as follows:

$$\delta = \frac{t_\gamma \cdot S}{\sqrt{n}}, \quad (2)$$

where n is the number of trials; S is the "corrected" standard deviation; t_γ is the Student's *t* -statistic.

3. The random variable X is distributed according to a law that differs from the normal distribution.

Under this condition, with a sufficiently large number of trials (n > 30) and reliability approximately equal to γ, the upper bound of the error can be calculated using formula (1), if the standard deviation σ of the random variable X is known; if the value of σ is unknown, its estimate S – the "corrected" standard deviation — can be substituted into formula (1), or formula (2) can be used.

Let's emphasize, that the larger the value of n, the smaller the difference between the results, given by both formulas. This is explained by the fact that as $n \to \infty$, the Student's distribution approaches the normal distribution.

From relations (1) and (2), it follows that the error of the Monte-Carlo method is inversely proportional to the square root of the number of trials. This characteristic will be used as the basis for further study of this method's error.

From the above description, it follows that the Monte-Carlo method is closely related to the tasks of probability theory, mathematical statistics, and computational mathematics. When considering the task of modeling random variables (especially uniformly distributed ones), number theory methods also play an important role in this case.

The given statements show that the convergence of the Monte-Carlo method is convergence in probability. This circumstance should hardly be considered disadvantage, as probabilistic methods largely justify themselves in practical applications. As for problems that have a probabilistic description, the convergence in probability is, to some extent, even natural when solving them.

Analysis of recent research and publications. Among numerical methods in mathematical calculations, there has been a recent resurgence of Monte-Carlo methods [1–4]. This name refers to a group of numerical methods based on obtaining a large number of realizations of a stochastic process, which is constructed in such way that its probabilistic characteristics coincide with the corresponding parameters of the problem at hand. Monte-Carlo methods are used for solving problems in industries of physics, mathematics, economics, optimization, control theory, and others. Fundamental works on Monte-Carlo methods appeared in 1955–1956. Since then, many scientific works have been written, describing the application of this method [5-10]. Even a superficial review of the titles of these works allows one to conclude that the Monte-Carlo method is applicable for solving practical problems in various fields of science and engineering.

The distinctive feature of Monte-Carlo methods is their experimental nature. For clarity, we will refer to a procedure as a Monte-Carlo method, if it involves the use of statistical sampling techniques for an approximate solution given mathematical or physical problem.

Monte-Carlo methods have had a significant impact on the development of computational mathematics and continue to do so, while in solving certain classes of problems, they successfully combine with other computational methods and complement them. Their application is particularly justified for problems that allow a probabilistic theoretical description. This is explained both by the naturalness of obtaining a response with a certain given probability in problems with probabilistic content, and by the significant simplification of the solution procedure. Currently, there are statistical methods for solving certain classes of partial differential equations, integral equations, eigenvalue problems, and systems of linear algebraic equations.

Among other computational methods, the Monte-Carlo method stands out for its simplicity and versatility. Slow convergence is a significant drawback of the method, but this article will describe its computational modifications, that ensure high convergence rates under

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

certain assumptions. However, the computational procedure becomes more complex, bringing it closer to other procedures in computational mathematics. The convergence of the Monte-Carlo method is probabilistic convergence. This fact should hardly be considered a disadvantage, as probabilistic methods are often highly effective in practical applications. The convergence of the Monte-Carlo method is probabilistic convergence. This circumstance should hardly be considered disadvantage, as probabilistic methods largely justify themselves in practical applications. As for problems with probabilistic descriptions, when solving them, convergence in probability is, to some extent, even natural. This explains the inherent possibility of parallelizing the method. Since individual Monte-Carlo iterations are typically independent of each other, they can be easily distributed across multiple threads or nodes in a cluster system. This makes the method ideal for parallel and distributed calculations.

Finally, it should be noted that the accuracy of the method's calculations heavily depends on the quality of the pseudo-random number generator used, while the speed of calculations is determined by the function describing the analyzed process and, of course, the performance of the "compute unit" itself. Today, the clock speed of modern processors has surpassed the GHz threshold, and the amount of random access memory in personal computers is measured in gigabytes. Considering that the relevant class of tasks will be processed using parallel computing, the performance of the "compute unit" is no longer a limiting factor (but rather a determining one) for the application of power-sensitive numerical algorithms in solving multidimensional problems. A practical illustration of the method's operational mechanism and some fundamental features of its implementation will be examined in the context of solving typical applied problems.

The purpose. The research aim is to improve understanding of the behavior of the Monte Carlo method in parallel computing and to identify optimal load distribution strategies to ensure fast and accurate convergence. Development and analysis of the efficiency of parallelizing the Monte-Carlo method is implemented using the *Python* programming language. Main focus is on studying the convergence of the method, examining the impact of parallelization on the accuracy and convergence speed, as well as assessing the scalability of computations on multi-core processors.

Main research material presentation. Many software applications developed for single-processor computing systems lack an adequate level of parallelism. Attempts to parallelize them at the level of individual loops often do not yield the desired results, leading to low code performance.

The Monte-Carlo method occupies a prominent position among different numerical methods in mathematical calculations. However, its use has been limited in the past due to the substantial computational resources it requires.

It is important to highlight, that the Monte-Carlo method allows finding an approximate solution to a problem at a specific fixed point, without requiring knowledge of the solutions at all points in the grid domain. This makes it notably different from, for instance, traditional methods of solving the Dirichlet problem.

The simplified computation scheme using the Monte-Carlo method is shown in Fig. 1.

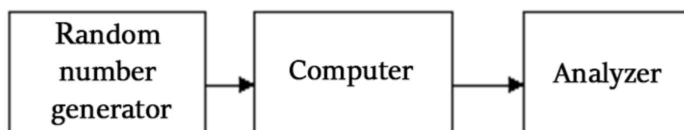

Figure 1 - Monte-Carlo method calculation scheme

The application of this method provides a new way to view the idea of parallelizing computations and using parallel computing technologies. As intermediate calculations can be carried out independently across different processor cores, their results can be consolidated on a dedicated "analyzer" core. This allows the parallel computing algorithm to be represented with a scheme, as illustrated in Fig. 2.

According to this scheme, a single random number generator assigns a specific random value to each "compute unit". However, this architecture has a significant drawback: the constant transmission of information through communication channels creates delays and reduces system's performance. Additionally, the quality of random number generator can affect the result, as repetitions or similar values may occur in the sequence of random numbers. However, these factors do not have a significant impact on the calculation error.

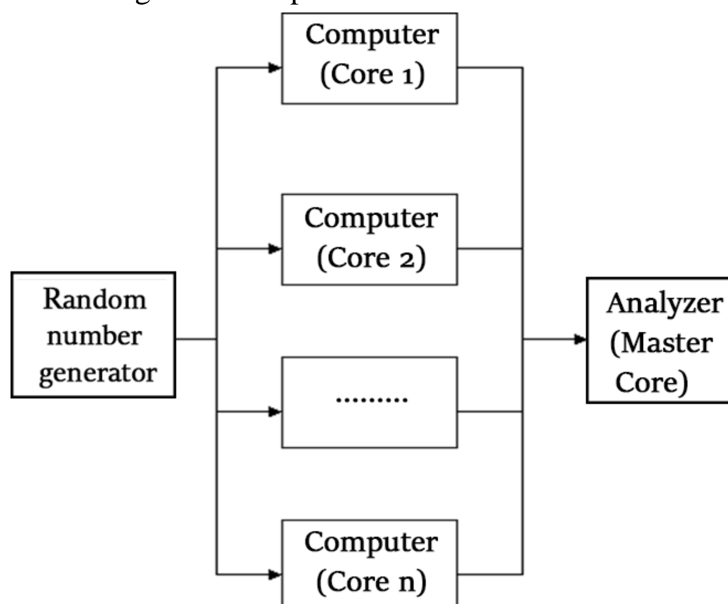

Figure 2 - Monte-Carlo parallel computing algorithm

The experience in developing parallel computations allowed to improve this scheme. The modified algorithm, shown in Fig. 3, assumes that each compute unit has a separate random number generator. This removes the necessity for constant data transmission between the generator and the processing cores, greatly accelerating the computation process.

Thus, Monte-Carlo algorithms demonstrate high resilience to changes in input data, provide maximum parallel efficiency, and minimize computation time. By appropriately distributing computing cores, it is possible to organize parallel execution of calculations across the entire domain, thereby improving the overall system performance.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

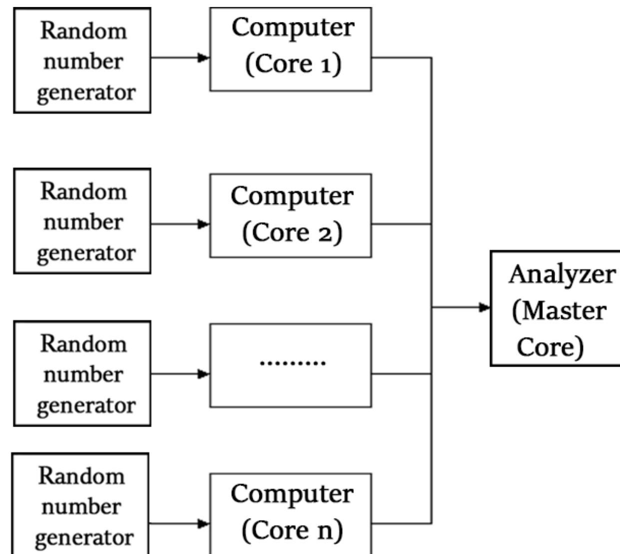

Figure 3 - Modified Monte-Carlo parallel computing algorithm

Realization of parallel computations was carried out using the *Python* programming language. It is worth mentioning that *Python* is currently one of the most widely used programming languages for developing and modeling parallel computations. This is attributed to its ease of use, extensive library ecosystem, and an active developer community, that continuously enhances tools and shares expertise. The use of *Python* for realization parallel computations is explained by several key factors, including ease of modeling, the availability of specialized libraries, a wide range of tools for handling large volumes of data, as well as integration with modern hardware accelerators such as graphics processing units (GPUs) and tensor processing units (TPUs).

The popularity of *Python* continues to grow rapidly alongside the development of parallel computing and artificial intelligence. This makes it an ideal option for both academic research and commercial use in the field of deep learning. A number of prominent companies, such as Google, Facebook, and OpenAI, extensively utilize *Python* in their developments, thereby reinforcing its dominance in the realms of artificial intelligence and parallel programming.

One of *Python`s* key advantages is its convenience for rapid prototyping. Due to the interpreted nature of the language, developers can swiftly experiment with various parallel computation architectures, adjust model parameters, and obtain real-time results, which significantly accelerates the development process.

In addition, *Python* is renowned for its high compatibility with other programming languages and frameworks. For instance, modules such as Cython facilitate seamless interaction with *C/C++*, thereby significantly enhancing computational performance. *Python* also integrates seamlessly with leading big data processing tools, such as *Apache Spark*, making it a versatile solution for machine learning and data analysis tasks.

Thus, *Python* remains the leading programming language for implementing parallel computations, offering exceptional flexibility, ease of use, and a powerful toolkit for handling complex computational tasks.

The software implementation of the computations was developed based on the principles of object-oriented programming. This approach is justified by the fact that object-oriented programming facilitates the creation of modular, reusable, and scalable parallel computations. Through encapsulation, inheritance, and polymorphism, efficient systems for multi-threaded and multi-processor data processing can be built. Therefore, a class was created to perform the necessary calculations:

```
class Integral:
def __init__(self, a, b, integrand, integral_mlt=1):
self.__a = None
self.__b = None
self.__integrand = None
self.__variables = None
self.__integral_mlt = None
```

The following libraries were also utilized, enabling efficient work with computations in both numerical and symbolic mathematics, as well as facilitating the implementation of multiprocessing programming:

```
import numpy as np
from sympy import sympify, zoo, SympifyError
from multiprocessing import Pool, cpu_count
from functools import partial
import time
```

NumPy was used as the primary library for working with multidimensional arrays and performing numerical computations. For the purpose of these studies, it supported array manipulation and the use of optimized functions for efficient calculations.

sympify() function from the symbolic computation library *SymPy* was used to convert the necessary data into a symbolic expression; the symbol *zoo* was employed to denote undefined values arising in mathematical expressions; the *SympifyError* class was utilized to handle errors occurring due to incorrect input in the *sympify()* function. These elements facilitate symbolic computations, enabling the efficient handling of mathematical expressions and exceptional cases.

Multiprocessing library was utilized for the implementation of parallel computations. *Pool* class in the *multiprocessing* module allows creation of a pool of processes for parallel task execution. It creates a specified number of processes and distributes tasks among them, which allows efficient utilization of multi-core processor resources.

cpu_count() function returns an integer representing the number of processor cores available for computation. In this research, it was utilized to determine the optimal number of processes for multiprocessing tasks. This function enables the automatic configuration of a process pool (*Pool*) to ensure efficient resource utilization. It has proven to be exceptionally useful when handling intensive computational workloads, such as machine learning, big data processing, and parallel algorithms.

partial function from the *functools* module enabled the creation of new functions based on existing ones by predefining certain arguments. This not only streamlined function calls, where specific arguments are frequently used, but also facilitated the passing of functions as

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

parameters in multithreaded code. Moreover, it contributed to the development of new function variations without redundant code duplication.

time library has provided the capability to work with time.

monte_carlo_method_parall method performs parallel computations by leveraging the multiprocessing library, effectively distributing the computational load among several processes:

```
    def monte_carlo_method_parall(n: int, integral: Integral,
**kwargs):
        cpu_cnt = cpu_count()
        if n <= cpu_cnt:
            chunks = [1] * n
        else:
            chunk_size = n // cpu_cnt
            remainder = n % cpu_cnt
            chunks = [chunk_size + 1 if i < remainder else
chunk_size for i in range(cpu_cnt)]
        monte_partial = partial(monte_carlo_method, inte-
gral=integral, **kwargs)
        with Pool(processes=cpu_cnt) as pool:
            results = pool.map(monte_partial, chunks)
        return np.mean(results)
```

At the first stage, method determines the number of available system cores, using the cpu_count() function. Then, if the number of cores is greater than or equal to the number of points to be generated, each process handles one point (if there are more points than cores, some processes may remain unused). If the number of points exceeds the available cores, method calculates cluster size – number of points each process should handle, by performing integer division of the total number of points by the number of cores. In cases where the points do not divide evenly, the remainder is distributed uniformly across the processes using the following expression:

```
    chunks = [chunk_size + 1 if i < remainder else chunk_size for i
in range(cpu_cnt)]
```

Then method utilizes the *partial* function to create a partial function, that incorporates the common arguments required for all processes to compute the integral values (specifically, the integral object and additional parameters passed through kwargs). This partial function is based on the *monte_carlo_method*.

```
    monte_partial = partial(Integrator.monte_carlo_method, inte-
gral=integral, **kwargs)
```

After the distribution of points, the *monte_carlo_method_parall* method utilizes the *Pool* from the multiprocessing module to initiate several parallel processes. *Pool* invokes the created partial function across multiple processes, passing it the corresponding number of points for processing. The results from each process are stored in the *results* variable:

```
    results = pool.map(monte_partial, chunks)
```

Once all processes are completed, method calculates the average of the obtained partial results using the *mean* function and returns the final outcome.

The initial data for the class of problems under study are wrapped in the condition *if __name__ == "__main__":*

```
if __name__ == "__main__":
    integral = Integral(0, 1, "(exp(-x)+1)**2")
    n = 60000
    start_time = time.time()
    result = monte_carlo_method_parall(n, integral)
    end_time = time.time()
    exec_time = end_time - start_time
```

Furthermore, to evaluate the efficiency of parallelization, the computation time for both sequential and parallel algorithms is determined. In the given case, an integral is computed for a function that lacks analytical quadrature solutions.

The efficiency of the parallel code was assessed by measuring its acceleration and parallelization effectiveness. These metrics were calculated using well-established formulas from parallel computing theory.

speedup attained by executing a parallel algorithm on n cores, in comparison to sequential execution, is determined by the following formula:

$$S(n)=\frac{t_1}{t_n}, \tag{3}$$

where *S* is the acceleration; t_1 is the time taken to solve the task on a scalar processor; t_n is the time taken to solve the task on a processor with n cores.

Therefore, acceleration is calculated as the ratio between the execution time of a task on a scalar processor and the time taken by the parallel algorithm. The parameter *n* serves to represent the computational complexity of the problem at hand.

The *efficiency* of parallelization is determined by the following formula:

$$E(n)=\frac{t_1}{t_n \cdot n}=\frac{S(n)}{n}. \tag{4}$$

For this type of computation, the effectiveness of parallelization represents the average duration of the algorithm's execution during which processor cores are actively involved in solving the task.

It should be noted that the control experiments were conducted using a computer with the following specifications:

Processor type: Core(TM) i5-11400H; clock frequency: 2.70GHz; number of cores: 6.

In accordance with expressions (3) and (4), the following efficiency estimates of the code parallelization were obtained: $S(n) = 4{,}469$; $E(n) = 0{,}745$.

Analysis of the conducted research revealed the following. Efforts to improve the quality of parallel computing based on one metric (such as speedup or efficiency) may lead to a deterioration in another metric, as the quality indicators of parallel computations are often conflicting. An increase in acceleration is typically achieved by increasing the number of processor cores, which, however, often results in a efficiency decrease. Conversely, enhanced efficiency is frequently attained by reducing the number of processor cores(with the ideal case of perfect efficiency, E(n) = 1, being effortlessly achieved by using a single-core processor.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

As a result, the development of parallel computation methods often involves selecting a certain compromise, taking into account the desired acceleration and efficiency metrics, as it demonstrated by the presented studies.

The peculiarities of studying the accuracy using the Monte-Carlo method were as follows. It is known that the accuracy of the Monte-Carlo method possesses several distinctive features. The initial test revealed, that the computational error is dependent on the number of tests, denoted as n. This indicates that as n increases, the obtained results approach the actual value, and the absolute error gradually diminishes. On average, the theoretical convergence rate of the Monte-Carlo method is determined as follows $O(1/\sqrt{n})$, meaning that the error decreases inversely proportional to the square root of the number of trials.

To verify this, nine series of approximations were conducted: first approximation involved 50 points, second 100 points, and so on up to 50,000 points. Each experiment was repeated 30 times for each set of points, after which the average absolute error was calculated. Since the Monte-Carlo method relies on randomly generated points, the error can vary depending on the quality of the random sampling. Therefore, to minimize the impact of randomness, each experiment was repeated multiple times, and the resulting outcomes were averaged. Based on this data, a graph of the mean absolute error was constructed for sample sizes of 50, 100, 500, 1000, 2000, 2500, 5000, 10,000, and 50,000 tests.

Fig. 4 illustrates the convergence of the Monte-Carlo method.

Let us analyze the obtained results. The graph illustrates the dependence between the absolute error and the number of points, n, in the Monte-Carlo method. Upon examining it, the following conclusions can be drawn:

− the typical convergence behavior $O(1/\sqrt{n})$ of the Monte-Carlo method is confirmed;

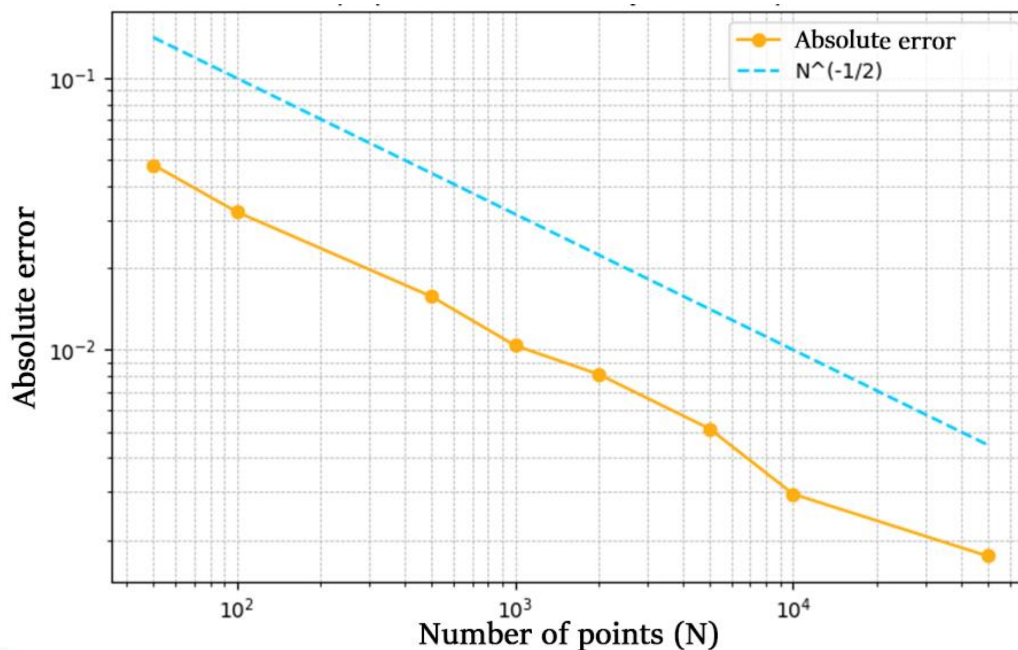

Figure 4 - Monte-Carlo convergence graph

− increasing n leads to a decrease in the error, albeit slowly, which is a characteristic feature of this method;

− the actual error aligns well with the theoretical estimate, indicating the correct implementation of the parallel algorithm;

− the x-axis (number of points) and y-axis (absolute error) are presented on a logarithmic scale, providing a clear depiction of the power-law relationship.

Fig. 5 presents a graph depicting the convergence approximation of the Monte-Carlo method.

An analysis of this dependency reveals, that for small values of n (up to 500–1000), significant deviations from the exact value occur, due to the high variance of the method when the number of samples is insufficient. In some cases, certain values may even fall outside the expected range, which is characteristic of the stochastic nature of the Monte-Carlo method.

At the same time, as the value of n increases, the method begins to stabilize. Starting from approximately 3000–4000 points, the results converge almost entirely with the exact value. This confirms the fundamental principle of Monte-Carlo: as the sample size grows, the method yields increasingly accurate results.

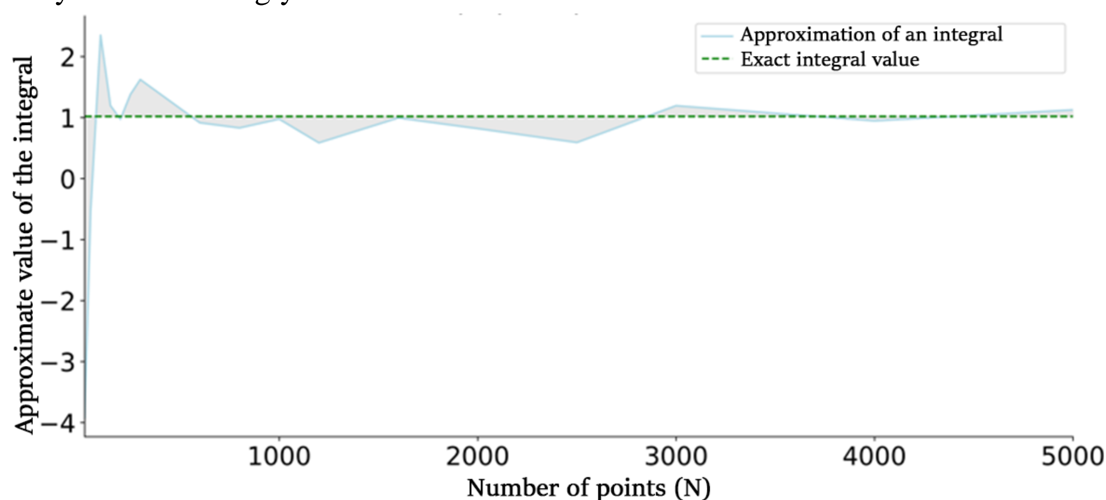

Figure 5 - The approximation convergence curve of the Monte-Carlo method

Conclusions. Research explores parallel technologies for modeling Monte-Carlo tasks. The relevance of the conducted studies is underscored by the fact that the Monte-Carlo method has had, and continues to have, a profound impact on the development of computational mathematics techniques. It has been shown that its use is especially appropriate for solving problems that can be described in theoretical and probabilistic terms. This is due to both the natural process of obtaining a solution with a given probability in probability-based tasks and the considerable simplification of the solution procedure.

This inherent ability to parallelize the method arises from the fact that individual Monte-Carlo iterations are generally independent of each other, making them easy to distribute across multiple threads or nodes within a cluster system. This characteristic makes the method especially well-suited for parallel and distributed computing. As a result, the main focus of

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

the research was to explore the intricacies of parallelizing computations in the context of solving a diverse array of applied problems.

It is worth noting that parallelizing the Monte-Carlo method is an important and relevant area in contemporary computational science, as it significantly enhances the speed and efficiency of calculations. With the availability of powerful processors and GPUs, this approach is widely applied across various fields of science and industry.

The calculation schemes that enhance productivity and performance are presented. The effectiveness of the proposed approach is demonstrated through research and graphical interpretations, showcasing the convergence and approximation of the developed method.

Finally, it should be highlighted that the Monte-Carlo method is a highly effective technique for addressing a broad spectrum of practical problems. It is easy to implement, and its error can be minimized by increasing the sample size.

Like any method, Monte-Carlo has its drawbacks. For instance, when working with a small amount of data, the variance becomes excessively high, and in such cases, it is often preferable to turn to alternative approaches.

REFERENCE

1. Rud O. Theoretical aspects of using the monte carlo method for modeling the evaluation of investment projects efficiency. *Market Infrastructure*. 2024. No. 79. URL: https://doi.org/10.32782/infrastruct79-24

2. Sirenko K. A., Mazur V. L., Derecha D. O. Application of the monte carlo method in charge calculations and regulation of the chemical composition of pig iron in the process of its smelting. *Casting processes*. 2023. Vol. 154, no. 4. P. 44–57. URL: https://doi.org/10.15407/plit2023.04.044

3. Nekrasova M. Monte-Carlo method and artificial intelligence: application of Monte-Carlo method in reinforcement learning. *Bulletin of the National Technical University «KhPI» Series: Dynamics and Strength of Machines*. 2024. No. 2. P. 47–52. URL: https://doi.org/10.20998/2078-9130.2024.2.315342

4. Velikova T., Mileva N., Naseva E. Method “Monte Carlo” in healthcare. *World Journal of Methodology*. 2024. Vol. 14, no. 3. URL: https://doi.org/10.5662/wjm.v14.i3.93930

5. Kroese D. P., Rubinstein R. Y. Simulation and the Monte Carlo Method. Wiley & Sons, Incorporated, John, 2016. 432 p.

6. Caflisch R. E. Monte Carlo and quasi-Monte Carlo methods. Acta Numerica. 1998. Vol. 7. P. 1–49. URL: https://doi.org/10.1017/s0962492900002804

7. Kroese D. P., Taimre T., Botev Z. I. Handbook of Monte Carlo Methods. Wiley & Sons, Incorporated, John, 2013. 772 p.

8. Binder K., Heermann D. W. Monte Carlo Simulation in Statistical Physics: An Introduction. Springer, 2019. 258 p.

9. Wang H. Monte Carlo Simulation with Applications to Finance. Taylor & Francis Group, 2012. 292 p.

10. Kalos M. H. Monte Carlo Methods in Quantum Problems. Dordrecht : Springer Netherlands, 1984. 291 p.

Received 02.04.2025.
Accepted 04.04.2025.

Розробка та дослідження паралельних технологій задач стохастичного програмування

У дослідженнях розглядаються паралельні технології моделювання задач методом Монте-Карло. Показано, що основна суть методу полягає у випадковому моделюванні великої кількості сценаріїв та статистичній обробці результатів, що пояснює природну можливість його розпаралелювання. Відзначається, що оскільки окремі ітерації методу Монте-Карло зазвичай незалежні одна від одної, їх легко розподілити між кількома потоками або вузлами кластерної системи. Це робить метод ідеальним для паралельних і розподілених обчислень. Приводяться схеми обчислень, які забезпечують збільшення продуктивності та швидкодії. Ефективність запропонованого підходу ілюструється дослідженнями та графічними інтерпретаціями збіжності та апроксимації розробленого підходу.

Швачич Геннадій Григорович – доктор технічних наук, професор кафедри програмного забезпечення комп'ютерних систем, НТУ «Дніпровська політехніка», Дніпро, Україна.

Щербина Павло Олександрович – асистент кафедри програмного забезпечення комп'ютерних систем, НТУ «Дніпровська політехніка», Дніпро, Україна.

Кабаченко Олег Вікторович – студент кафедри програмного забезпечення комп'ютерних систем, НТУ «Дніпровська політехніка», Дніпро, Україна.

Олішевський Ілля Геннадійович – доктор PhD, доцент кафедри безпеки інформації та телекомунікацій, НТУ «Дніпровська політехніка», Дніпро, Україна.

Іщук Павло Олександрович – аспірант кафедри програмного забезпечення комп'ютерних систем, НТУ «Дніпровська політехніка», Дніпро, Україна.

Shvachych Gennady – Doctor of science, Professor, Department of Computer Systems Software, Dnipro University of Technology, Dnipro, Ukraine.

Shcherbyna Pavlo – Assistant, Department of Computer Systems Software, Dnipro University of Technology, Dnipro, Ukraine.

Kabachenko Oleg – student of the Department of Computer Systems Software, Dnipro University of Technology, Dnipro, Ukraine.

Olishevskyi Ilya – Doctor PhD, assosiate professor, Department of Information Security and Telecommunications, Dnipro University of Technology, Dnipro, Ukraine.

Moroz Dmytro – Doctor PhD, assosiate professor, Department of Computer Systems Software, Dnipro University of Technology, Dnipro, Ukraine.

Ishchuk Pavlo – PhD student of the Department of Computer Systems Software, Dnipro University of Technology, Dnipro, Ukraine.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)