UDC 004.93

OPTIMIZATION OF PERMANENT DECOMPOSITION PROCEDURES USING PARALLELIZATION ALGORITHM

Turbal Y. V. – Dr. Sc., Professor, Head of the Department of computer science and applied mathematics, National University of Water and Environmental Engineering, Rivne, Ukraine.

Moroziuk A. Y. – Post-graduate student of the Department of computer science and applied mathematics, National University of Water and Environmental Engineering, Rivne, Ukraine.

ABSTRACT

Context. The problem of efficiently finding all permutations of a list of *N* elements is a key problem in many areas of computer science, such as combinatorics, optimization, cryptography, and machine learning. The object of the study was to analyze the procedure of permanent decomposition and propose an algorithm for its parallelization using modern features of working with threads in C#.

Objective. The goal of the work is the creation of the algorithm for parallelizing the generation of permutations using permanent decomposition processes.

Method. The main research method is the comparison of various algorithms with proposed parallelized algorithm, taking into account such criteria as accuracy and speed. Scientific works [10, 9, 17] present algorithms, including the regular permanent decomposition algorithm and the Johnson-Trotter algorithm. The Johnson-Trotter's algorithm is one of the most effective, so it has been taken as some kind of benchmark.

It is worth mentioning that each paralleling process has its own disadvantages, including additional resources needed fot data synchronization between threads. This can be minimized using both technical abilities of modern programming languages and optimization of the algorithm itself.

Results. The developed parallelized algorithm have improved performance of the regular permanent decomposition algorithm for solving the problem of finding all permutations.

Conclusions. The conducted experiments have confirmed the proposed parallelized algorithm's version is better from a performance standpoint than the regular one. The prospects for further research may include the application of the parallelized version of the algorithm to some practical tasks and comparison of the results.

KEYWORDS: parallelization, permanent decomposition, multi thread, algorithm, C#.

ABBREVIATIONS

TPL is a Task Parallel Library.

NOMENCLATURE

A is an input data list;

a_i is an i-th element of the input data list;

B is an initial list for parallel algorithm;

N is a number of elements in the input data list;

$O()$ is an algorithm's complexity;

$perm()$ is a permanent of the matrix;

S_N is a set of all permutations of the indices of the rows or columns of a matrix;

x is a number of all permutations for an initial list for parallel algorithm;

X_0 is a list of all permutations for an initial list for parallel algorithm;

X_j is an extended list of all permutations.

INTRODUCTION

In the modern theory of combinatorics and object generation algorithms, methods that allow solving complex problems, including permutations, combinations, and other structures, are of great importance. One of these technologies is the method of permanent decomposition, which has proven to be effective [17]. However, the process of generating combinatorial objects is a difficult task, especially for large matrices, since the number of possible permutations of matrix elements grows exponentially with its size (complexity $O(N!)$ [3]). Parallelization of computations can significantly increase the efficiency of this process. In this article, we will consider approaches to parallelizing the permanent decomposition algorithm and the main technical aspects of this process.

The object of study is the problem of generating all permutations of a list of *N* elements.

This is a fundamental problem in computer science with applications in areas such as combinatorics, optimization, cryptography, and algorithm design. Therefore, to improve the speed of the permanent decomposition algorithm it is possible to parallelize it and make some performance measurements.

The subject of study is the parallelization techniques that can be applied to permanent decomposition algorithm.

For the effectiveness comparison Johnson-Trotter and regular (sequential) version of algorithm will be used.

The purpose of the work is to implement a parallel version of the algorithm and compare performance with other algorithms.

1 PROBLEM STATEMENT

Suppose given the original list of N elements $A = [a_1, a_2, ..., a_N]$.

DOI 10.15588/1607-3274-2025-4-6

For a given list *A* the problem of generating all permutations can be divided into pieces and parallelized, to improve execution speed for this process using modern multi-core computers.

In turn, the problem of parallelizing permanent decomposition process is to implement algorithm of dividing original list into sub-lists and combine results from each thread/operation.

2 REVIEW OF THE LITERATURE

Sequential algorithms for generating permutations have been widely studied. One of the earliest and most notable methods is the Johnson-Trotter algorithm, introduced by Johnson [9] and Trotter [16]. This algorithm generates permutations using adjacent transpositions and is efficient in terms of memory usage. However, its computational complexity becomes a limiting factor for large values of *n*. Another significant contribution is Heap's algorithm, proposed by Heap [6], which generates permutations in a non-lexicographic order and is celebrated for its simplicity and in-place nature. Knuth, in The Art of Computer Programming [10], provides an extensive analysis of these and other permutation generation algorithms, highlighting their theoretical foundations and practical limitations.

With the rise of multi-core and distributed computing systems, researchers have turned their attention to parallel algorithms to enhance the efficiency of combinatorial generation. Akl, in The Design and Analysis of Parallel Algorithms [1], discusses the potential of parallel computing for solving combinatorial problems. More recently, in Structured Parallel Programming [12], author emphasize the importance of leveraging modern threading models and parallel frameworks to optimize computationally intensive tasks.

Permanent decomposition is a mathematical technique with applications in combinatorial optimization and matrix theory. Valiant, in his seminal work on the complexity of computing the permanent [18], highlighted its relevance in graph theory and statistical physics. Glynn [4] later proposed an efficient algorithm for computing the permanent of a matrix, which has been adapted for combinatorial generation tasks. Despite these advancements, the parallelization of permanent decomposition procedures remains an understudied area, with limited research addressing its implementation in multi-core systems.

Modern programming languages, such as C#, offer robust support for parallel computing through threading libraries and task-based parallelism. Richter, in his book "CLR via C#" [13], explores the threading capabilities of C# and their application to computationally intensive tasks. Similarly, Toub, in his lecture [11], provides practical insights into designing scalable parallel algorithms using the TPL [15]. These resources have been instrumental in developing efficient parallel implementations of combinatorial algorithms.

The generation of permutations and combinations has numerous applications across various domains. Garey and Johnson, in Computers and Intractability [8], discuss the role of combinatorial algorithms in solving NP-hard problems. In bioinformatics, Gusfield [5], highlights the use of combinatorial generation for sequence alignment and genome analysis. Babych and Turbal [17] show how the permanent decomposition methods might be applied to solve schedule generation problems.

3 MATERIALS AND METHODS

The permanent of a square matrix is a generalization of the determinant without taking into account the change of permutation signs. The algorithmic complexity of calculating the permanent grows exponentially with the size of the matrix, which limits its practical application for large matrices. The standard approach, based on trying all possible permutations of rows or columns, has a complexity of $O(N!)$. This makes the task inefficient for large matrices. To speed up the computation, efficient parallelization algorithms are needed.

The permanent of a matrix A of size $N \times N$ is defined as follows:

$$perm(A) = \sum_{\sigma \in S_N} \prod_{i=1}^{N} a_{i,\sigma(i)}. \quad (1)$$

Due to the absence of sign changes (as in the determinant), the constant has no analytically simple ways to compute it, so computing the constant is a combinatorially difficult task.

The method of permanent decomposition is used to efficiently generate combinatorial objects such as permutations, combinations with and without repetitions, and various subsets [17]. The main idea of the method is to perform a recursive process with gradual memorization of element indices instead of directly calculating the algebraic permanent of the incidence matrix. This reduces the number of operations and optimizes the generation process.

An example is the generation of permutations of the set {1, 2, 3}. Instead of checking all possible variants, the permanent decomposition method uses a special incidence matrix to build permutations step by step, which reduces computational complexity. As a result, the method offers an effective alternative to classical approaches such as the lexicographic method or the Johnson-Trotter algorithm.

This approach is especially useful in problems with a large number of objects or complex constraints, such as planning or resource optimization, where it is necessary to compute many possible options in a short time.

Parallelization of computations can significantly speed up the process of calculating a constant, especially when there are a large number of permutations or subtasks. The main approach to parallelization is based on dividing the computation of a constant into several independent threads or processes that can run simultaneously on different processor cores or cluster nodes.

Distribution of permutations: Since a persistent depends on all possible permutations of rows or columns, these permutations can be distributed among multiple

DOI 10.15588/1607-3274-2025-4-6

threads. Each thread computes its own part of the permutations, and then the results are combined. This approach allows you to linearly reduce the computation time in the case of perfect parallelization.

Recursive decomposition: Another parallelization approach is based on recursive decomposition of the matrix into smaller blocks. Each block can be computed independently, and these computations can be performed in parallel on different threads or nodes. For example, if we have a matrix of size $N \times N$, it can be divided into four blocks of size $N/2 \times N/2$, and the permanent of each of these blocks can be calculated on a separate thread.

The software implementation was made using the C# programming language, as it provides a wide range of tools for organizing parallel computing through the TPL [15], which allows for the efficient use of multi-core processors. The main tools for implementing parallel execution are the Parallel, Task classes [14] and asynchronous programming concepts.

One of the easiest ways to parallelize the calculation of a constant is to use the Parallel.For loop [2], which allows you to distribute tasks among several threads. In the context of calculating a permanent, each thread can be responsible for calculating the permanent for a certain subset of matrix permutations.

An alternative approach is the use of tasks, which allows for more flexible thread management and asynchronous execution of tasks, which makes the implementation of this approach more complex. It is worth noting that it is precisely because of the additional flexibility and the possibility of fine-tuning that the choice fell on this method of working with parallelism.

As already mentioned, parallelization should speed up the work by dividing it among several performers, while synchronization, resource sharing, and blocking [7] can degrade the performance of the software implementation of the algorithm, so it is important to maintain a balance between parallelization and the excessive costs of such a system. To implement the algorithm, we used both single-threading [2] and multi-threading [7].

The proposed algorithm is based on a combination of several approaches: permutation distribution and recursive decomposition. Take for example a list of N values (for better visualization, it can be a list of numbers from *1* to N). The purpose of a parallelized algorithm is to perform part of the work in parallel, while obtaining a list of all possible permutations for the input list. The algorithm itself in this case is as follows:

Here is the description of the algorithm:

1. The input is a list A of N elements $A = [a_1, a_2, ..., a_N]$.

2. Generate the initial list B, which will contain $(N-1)$ elements, for example, we take all the elements of the list except the first one – a_1.

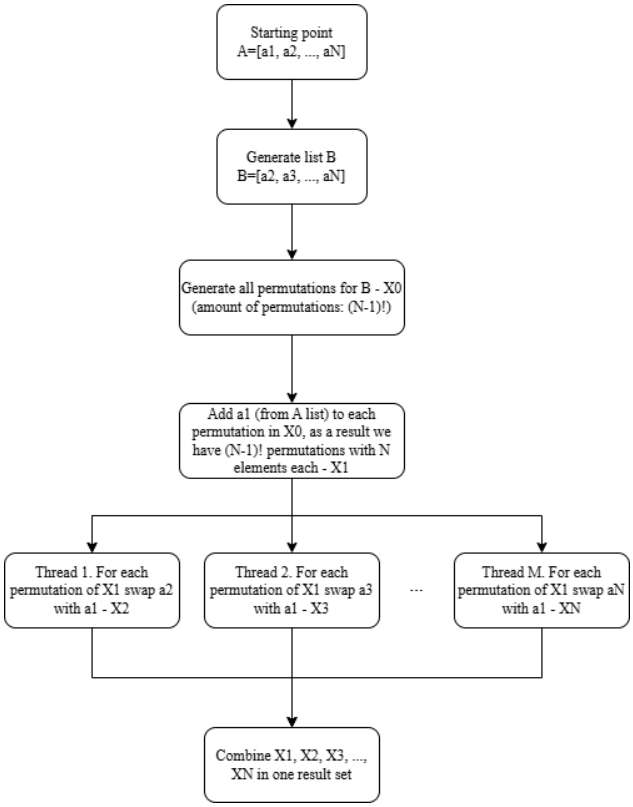

Figure 1 – Parallel permanent decomposition algorithm

3. Generate all possible permutations X_0 for list B in the main thread. As a result of this step, we will get x possible permutations, where $x = (N-1)!$, and the number of elements in each permutation is $(N-1)$.

4. Generate X_1 by adding one more element (a_1) to each permutation, as a result we get x permutations, but with the number of elements N in each.

5. Run a loop through our list A and process each iteration of this loop in parallel, running step 6 in a separate thread.

6. Generate a new list of combinations based on list X_1, swapping a_i and a_1.

7. After all parallel tasks are completed, combine the results, and the output is a list of all possible combinations of elements of list A.

The software implementation was made using the tools of the C# programming language. Main method that contains logic of running tasks in different threads is provided on Figure 2.

DOI 10.15588/1607-3274-2025-4-6

```
public static async Task<List<List<int>>> Get(int n, int numberOfThreads = 4)
{
    List<int> nums = new List<int>();
    for (int i = 1; i <= n; i++)
    {
        nums.Add(i);
    }
    List<List<int>> permutations = new List<List<int>>();

    var numsToBeProcessed = nums.Except(new List<int> { nums[0] }).ToList();
    var initialSet = Permute(numsToBeProcessed.ToArray());
    initialSet.ForEach(item => item.Add(nums[0]));
    permutations.AddRange(initialSet);
    List<Task> tasks = new List<Task>();

    foreach (var number in nums)
    {
        if (number != nums[0])
            tasks.Add(Task.Run(() => ProcessOneElement(nums, number, initialSet, nums[0])));
    }

    await Task.WhenAll(tasks);
    foreach (Task<List<List<int>>> task in tasks)
        permutations.AddRange(task.Result);

    return permutations;
}
```

Figure 2 – Main method of parallel algorithm implementation

4 EXPERIMENTS

The experimental study compared the performance of three approaches to finding all permutations of a set of *N* elements:

– Johnson-Trotter method;

– Permanent decomposition method (sequential variant);

– Permanent decomposition method (parallel variant).

The tests were conducted for different values of *N* (from 9 to 11, since the difference for less than 9 is relatively small and is not representative) on the same hardware using a multi-core processor for parallel execution. The experiments were performed on a computer with the following characteristics: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30GHz, 16 GB of RAM.

5 RESULTS

The fragment of the results of conducted experiments is presented in the Table 1.

The main results of the experiments show that the Johnson-Trotter method demonstrates the best results among the other tested algorithms, but as *N* grows, its execution time increases exponentially. The sequential variant of the permanent decomposition method also has an exponential increase in execution time, but it is slightly slower. The parallel variant of this method shows a reduction in execution time compared to sequential approaches, but as *N* grows, the resource consumption for thread synchronization increases.

In terms of memory consumption, the Johnson-Trotter method requires the least resources due to the minimum number of additional structures. Instead, the permanent decomposition methods, especially the parallel one, require more memory to store data on subtasks and control threads.

The efficiency of parallel execution is highest when using 4–8 cores, after which the performance gain decreases due to uneven distribution of tasks and synchronization overhead.

In terms of implementation complexity, the Johnson-Trotter method is the easiest to implement. On the contrary, permanent decomposition methods require a significant amount of code to handle recursions and synchronize threads in parallel execution.

Table 1 – The fragment of experimental results for selected methods, grouped by number of input elements

Number of elements (*n*)	Permanent decomposition method	Parallelized permanent decomposition method	Johnson-Trotter method
9	0.141s	0.138s	0.129s
10	1.12s	1.065s	0.987s
11	11.825s	11.004s	10.031s

6 DISCUSSION

As it evident from the Table 1, with the increasing of examples elements the complexity of the task is increased significantly.

The experimental results presented in this study provide valuable insights into the performance, scalability, and practicality of the Johnson-Trotter method and permanent decomposition methods for generating permutations. The findings highlight the trade-offs between execution time, memory consumption, and implementation complexity, offering a comprehensive understanding of the strengths and limitations of each approach.

The Johnson-Trotter method demonstrated superior performance in terms of execution time for smaller values of *N*, outperforming both sequential and parallel variants of the permanent decomposition method. This aligns with existing literature, which emphasizes the efficiency of the Johnson-Trotter algorithm for small to moderately sized problems [9].

The parallel variant of the permanent decomposition method showed a reduction in execution time compared to its sequential counterpart, particularly when utilizing 4-8 cores. This is consistent with the findings of McCool [12], who emphasize the potential of parallel computing for optimizing computationally intensive tasks. However, the performance gain diminishes as the number of cores increases beyond this range, primarily due to the overhead associated with thread synchronization and uneven task distribution. This observation highlights the challenges of scaling parallel algorithms for combinatorial problems, as noted by Akl [1].

DOI 10.15588/1607-3274-2025-4-6

CONCLUSIONS

Parallelization of the permanent decomposition algorithm is an important step to speed up the computation of large matrices. Thanks to modern parallel programming tools, it is possible to reduce the execution time of such algorithms by distributing computational tasks over several threads. The effectiveness of parallelization depends on the structure of the matrix, the chosen decomposition method, the uniformity of load distribution between threads, and the degree of parallelization.

The above algorithm works correctly and has prospects for further use for applied tasks.

The scientific novelty of obtained results is that the algorithm of parallelization of the permanent decomposition method is firstly proposed. It allows us to improve the efficiency of this method and make it more scalable for modern multi-core systems.

The practical significance of obtained results is that the software implementation of proposed algorithm is developed, as well as experiments to study their efficiency are conducted. The experimental results allow to recommend the proposed algorithm for use in practice to gain better results.

Prospects for further research are to improve load balancing methods for parallel algorithms; to adapt and apply developed solution to distributed computing environments.

REFERENCES

1. Akl S. G. The design and analysis of parallel algorithms. Englewood Cliffs, N.J : Prentice-Hall, 1989, 401 p.
2. Albahari J., Albahari B. C# 7.0 in a nutshell: the definitive reference. [S. l.], O'Reilly Media, 2017, 1088 p.
3. Cormen T. H., Stein C. , Rivest R. L. Introduction to algorithms. [S. l.], MIT Press, 2009, 1320 p.
4. Glynn D. G. The permanent of a square matrix [Electronic resource], *European journal of combinatorics*, 2010, Vol. 31, No. 7, pp. 1887–1891. Mode of access: https://doi.org/10.1016/j.ejc.2010.01.010
5. Gusfield D. Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge [England], Cambridge University Press, 1999, 534 p.
6. Heap B. R. Permutations by interchanges [Electronic resource], *The computer journal,* 1963, Vol. 6, No. 3, pp. 293–298. Mode of access: https://doi.org/10.1093/comjnl/6.3.293
7. Herlihy M., Shavit N. Art of multiprocessor programming . [S. l.], Elsevier Science & Technology Books, 2011.
8. Johnson D. S., Garey. M. R. Computers and intractability: A guide to the theory of NP-completeness. New York, W.H. Freeman, 1983, 340 p.
9. Johnson S. M. Generation of permutations by adjacent transposition [Electronic resource], *Mathematics of computation,* 1963, Vol. 17, No. 83, pp. 282–285. Mode of access: https://doi.org/10.1090/s0025-5718-1963-0159764-2
10. Knuth D. E. The art of computer programming: generating all tuples and permutations. 2nd ed. Upper Saddle River [etc.], Addison-Wesley, 2009, 128 p.
11. Let's talk parallel programming with Stephen Toub [Electronic resource], *Microsoft Learn*. Mode of access: https://learn.microsoft.com/en-us/shows/on-dotnet/lets-talk-parallel-programming-with-stephen-toub
12. McCool M., Reinders J., Robison A. Structured parallel programming: patterns for efficient computation. [S. l.], Elsevier Science & Technology Books, 2012, 432 p.
13. Richter J. CLR via C# . [S. l.] : Microsoft Press, 2012, 894 p.
14. Task-based asynchronous programming – .NET [Electronic resource], *Microsoft Learn*. Mode of access: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming
15. Task parallel library (TPL) – .NET [Electronic resource] // Microsoft Learn. Mode of access: https://learn.microsoft.com/uk-ua/dotnet/standard/parallel-programming/task-parallel-library-tpl
16. Trotter H. F. Algorithm 115: perm [Electronic resource], *Communications of the ACM,* 1962, Vol. 5, No. 8, pp. 434–435. Mode of access: https://doi.org/10.1145/368637.368660
17. Turbal Y. V., Babych S. V., Kunanets N. E. Permanent decomposition algorithm for the combinatorial objects generation [Electronic resource], *Radio electronics, computer science, control*, 2022, No. 4, P. 119. Mode of access: https://doi.org/10.15588/1607-3274-2022-4-10
18. Valiant L. G. The complexity of computing the permanent [Electronic resource], *Theoretical computer science*, 1979, Vol. 8, No. 2, pp. 189–201. Mode of access: https://doi.org/10.1016/0304-3975(79)90044-6

Received 19.05.2025.
Accepted 19.09.2025.

DOI 10.15588/1607-3274-2025-4-6

УДК 004.93

ОПТИМІЗАЦІЯ ПРОЦЕДУР ДЕКОМПОЗИЦІЇ ПЕРМАНЕНТУ З ВИКОРИСТАННЯМ АЛГОРИТМУ РОЗПАРАЛЕЛЮВАННЯ

Турбал Ю. В. – д-р техн. наук, професор, завідувач кафедри кафедри комп'ютерних наук та прикладної математики Національного університету водного господарства та природокористування, Рівне, Україна.

Морозюк А. Ю. – аспірант кафедри комп'ютерних наук та прикладної математики Національного університету водного господарства та природокористування, Рівне, Україна.

АНОТАЦІЯ

Актуальність. Задача ефективного знаходження всіх перестановок списку з N елементів є ключовою проблемою в багатьох областях комп'ютерних наук, таких як комбінаторика, оптимізація, криптографія та машинне навчання. Мета дослідження – проаналізувати процедуру перманентної декомпозиції та запропонувати алгоритм для її розпаралелювання з використанням сучасних можливостей роботи з потоками в мові C#.

Мета роботи – метою роботи є створення алгоритму для розпаралелювання генерації перестановок з використанням перманентних процесів декомпозиції.

Метод. Основним методом дослідження є порівняння різних алгоритмів із запропонованим розпаралелеленим алгоритмом з урахуванням таких критеріїв, як точність та швидкість. У наукових працях [10, 9, 17] представлено алгоритми, серед яких регулярний алгоритм перманентної декомпозиції та алгоритм Джонсона-Троттера. Алгоритм Джонсона-Троттера є одним з найефективніших, тому його було взято за певний еталон.

Варто зазначити, що кожен процес розпаралелювання має свої недоліки, зокрема, додаткові ресурси, необхідні для синхронізації даних між потоками. Це можна мінімізувати, використовуючи як технічні можливості сучасних мов програмування, так і оптимізацію самого алгоритму.

Результати. Розроблений розпаралелений алгоритм дозволив покращити продуктивність звичайного алгоритму перманентної декомпозиції для розв'язання задачі знаходження всіх перестановок.

Висновки. Проведені експерименти підтвердили, що запропонована розпаралелена версія алгоритму є кращою з точки зору продуктивності, ніж звичайна. Перспективами подальших досліджень може бути застосування розпаралеленої версії алгоритму до деяких практичних задач та порівняння отриманих результатів.

КЛЮЧОВІ СЛОВА: розпаралелювання, постійна декомпозиція, багатопотоковість, алгоритм, C#.

ЛІТЕРАТУРА

1. Akl S. G. The design and analysis of parallel algorithms / Selim G. Akl. – Englewood Cliffs, N.J : Prentice-Hall, 1989. – 401 p.
2. Albahari J. C# 7.0 in a nutshell: the definitive reference / J. Albahari, B. Albahari. – [S. l.] : O'Reilly Media, 2017. – 1088 p.
3. Cormen T. H. Introduction to algorithms / Thomas H. Cormen, Clifford Stein, Ronald L. Rivest. – [S. l.] : MIT Press, 2009. – 1320 p.
4. Glynn D. G. The permanent of a square matrix [Electronic resource] / David G. Glynn // European journal of combinatorics. – 2010. – Vol. 31, no. 7. – P. 1887–1891. – Mode of access: https://doi.org/10.1016/j.ejc.2010.01.010
5. Gusfield D. Algorithms on strings, trees, and sequences: computer science and computational biology / Dan Gusfield. – Cambridge [England] : Cambridge University Press, 1999. – 534 p.
6. Heap B. R. Permutations by interchanges [Electronic resource] / B. R. Heap // The computer journal. – 1963. – Vol. 6, No. 3. – P. 293–298. – Mode of access: https://doi.org/10.1093/comjnl/6.3.293
7. Herlihy M. Art of multiprocessor programming / Maurice Herlihy, Nir Shavit. – [S. l.] : Elsevier Science & Technology Books, 2011.
8. Johnson D. S. Computers and intractability: A guide to the theory of NP-completeness / David S. Johnson, M. R. Garey. – New York : W. H. Freeman, 1983. – 340 p.
9. Johnson S. M. Generation of permutations by adjacent transposition [Electronic resource] / Selmer M. Johnson // Mathematics of computation. – 1963. – Vol. 17, No. 83. – P. 282–285. – Mode of access: https://doi.org/10.1090/s0025-5718-1963-0159764-2
10. Knuth D. E. The art of computer programming: generating all tuples and permutations / Donald Ervin Knuth. – 2nd ed. – Upper Saddle River [etc.] : Addison-Wesley, 2009. – 128 p.
11. Let's talk parallel programming with Stephen Toub [Electronic resource] // Microsoft Learn. – Mode of access: https://learn.microsoft.com/en-us/shows/on-dotnet/lets-talk-parallel-programming-with-stephen-toub
12. McCool M. Structured parallel programming: patterns for efficient computation / Michael McCool, James Reinders, Arch Robison. – [S. l.] : Elsevier Science & Technology Books, 2012. – 432 p.
13. Richter J. CLR via C# / Jeffrey Richter. – [S. l.] : Microsoft Press, 2012. – 894 p.
14. Task-based asynchronous programming – .NET [Electronic resource] // Microsoft Learn. – Mode of access: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-based-asynchronous-programming
15. Task parallel library (TPL) – .NET [Electronic resource] // Microsoft Learn. – Mode of access: https://learn.microsoft.com/uk-ua/dotnet/standard/parallel-programming/task-parallel-library-tpl
16. Trotter H. F. Algorithm 115: perm [Electronic resource] / H. F. Trotter // Communications of the ACM. – 1962. – Vol. 5, no. 8. – P. 434–435. – Mode of access: https://doi.org/10.1145/368637.368660
17. Turbal Y. V. Permanent decomposition algorithm for the combinatorial objects generation [Electronic resource] / Y. V. Turbal, S. V. Babych, N. E. Kunanets // Radio electronics, computer science, control. – 2022. – No. 4. – P. 119. – Mode of access: https://doi.org/10.15588/1607-3274-2022-4-10
18. Valiant L. G. The complexity of computing the permanent [Electronic resource] / L. G. Valiant // Theoretical computer science. – 1979. – Vol. 8, no. 2. – P. 189–201. – Mode of access: https://doi.org/10.1016/0304-3975(79)90044-6

DOI 10.15588/1607-3274-2025-4-6