The method of parallelization of dynamic sequential system programs using networks of multifaceted processes ⋆

Oleg Savenko[1,*,†], Mykola Fedula[1,†], Jan Rabcan[2,†], Kyrylo Voznyi[1,†], Serhii Mostovyi[1,†] and Anatolii Barabash[1,†]

[1] *Khmelnytskyi National University, Instytutska street 11, 29016, Khmelnytskyi, Ukraine*
[2] *Zilina University, Univerzitná 8215, 010 26 Žilina, Slovakia*

Abstract

This article examines the unique characteristics of dynamic sequential system programs and the challenges in their effective parallelization. The complexity of automatically parallelizing these programs stems from intricate data dependencies, irregular computation structures, dynamic task generation, and the need to synchronize access to shared resources. These factors limit the effectiveness of traditional parallel execution methods and highlight the need for specialized techniques.

To address these challenges, a novel method for parallelizing dynamic sequential system programs is introduced, utilizing networks of interacting processes. This approach formalizes the computation structure, allowing for the decomposition of programs into independent or partially independent segments. By organizing the execution of these segments as a system of collaborative processes, the method enhances resource utilization and improves scalability.

The article also presents a model representing a sequential system program as a network of interacting processes, outlining the mechanisms for interaction and synchronization. These mechanisms ensure proper data exchange, maintain execution consistency, and prevent conflicts in accessing shared resources, thereby facilitating efficient parallel execution on multi-core systems.

A study evaluating the effectiveness of the proposed method confirms its benefits, demonstrating improved productivity in executing dynamic system programs. This approach not only optimizes computing processes but also enhances hardware resource utilization. In the cybersecurity domain, it provides a computational basis for parallel synthesis and evaluation of adaptive honeypots and decoy networks, particularly in scenarios modeled on game-theoretic principles.

Additionally, the proposed method includes a deadlock-aware solver as an auxiliary safety-verification stage. The solver is applied after a multifaceted process network has been formed and before its mapping onto computational resources. It uses process signatures, resource-state analysis, fuzzy clustering, and fuzzy inference to estimate the probability of cyclic resource waiting and to recommend corrective actions when the planned parallel structure becomes unsafe.

Future research will explore the application of this method for parallelizing processes within distributed computing environments.

Keywords

distributed system, computer system, parallelization, processes, system programs, cyber deception, deadlock forecasting model, process signature, resource-interaction solver [1]

1. Introduction

The parallelization of dynamic sequential system programs using multifaceted process networks is a relevant research area in parallel computing and software optimization. This relevance is determined by current trends in the development of computing technology. Most modern computer systems are based on multicore processors and distributed computing environments, which provide significant potential for parallel program execution. However, a significant part of

⋆ *IntelITSIS-2026: 7th International Workshop on Intelligent Information Technologies & Systems of Information Security, July 3, 2026, Khmelnytskyi, Ukraine*
[1,*] Corresponding author.
[†] These authors contributed equally.
savenko_oleg_st@ukr.net (O. Savenko); mailfm2000@gmail.com (M. Fedula); jan.rabcan@fri.uniza.sk (J. Rabcan); k.vozniy@gmail.com (K. Voznyi); serhii.mostovyi@khmnu.edu.ua (S. Mostovyi) wkondrael@gmail.com (A. Barabash)
0000-0002-4104-745X (O. Savenko); 0000-0002-3765-2016 (M. Fedula); 0000-0003-2835-9114 (J. Rabcan); 0009-0007-2545-565X (K. Voznyi); 0000-0002-9505-3206 (S.Mostovyi); 0009-0000-9607-1775 (A. Barabash)

existing software, particularly system-level software, was developed as sequential, that is, oriented toward the execution of instructions within a single thread. As a result, the hardware capabilities of modern systems are not fully utilized, which reduces the overall efficiency of computation. Therefore, there is a need to study methods that make it possible to transform sequential programs into parallel ones without radically changing their structure or completely rewriting the program code.

The relevance of this topic is especially evident in the case of dynamic system programs whose operation depends on the system state, the data being processed, and the execution conditions. Unlike static algorithms, where the structure of computation is determined in advance, dynamic programs form the execution flow during operation. The presence of conditional branches, complex data structures, and variable dependencies between operations significantly complicates the process of automatic parallelization. Under such conditions, traditional optimization methods often prove insufficiently effective, which justifies the need to search for new approaches to organizing parallel computation.

One promising approach is the use of the multifaceted process network model to represent the structure of a program. Within this approach, a sequential program is considered as a system of interacting processes, each of which performs a separate part of the computation and exchanges data with other components. Such a model makes it possible to represent the program in the form of a dependency graph, which significantly facilitates the analysis of the computational structure and the identification of fragments that can be executed in parallel. In this case, a multifaceted process acts as a universal computational element capable of simultaneously performing computational operations, exchanging data, and ensuring synchronization with other processes.

The use of a multifaceted process network creates an opportunity to formalize the structure of complex software systems and carry out their further optimization. Representing a program as a set of interconnected processes makes it possible to determine independent computational sections and organize their parallel execution. This, in turn, contributes to improving software performance and to the more efficient use of the computational resources of modern multicore and distributed systems. One possible application area of such models is cybersecurity, where defensive systems often require the analysis of many alternative configurations under dynamic conditions. For example, the synthesis of honeypot and decoy-network structures may involve evaluating multiple attacker paths, defensive responses, and resource allocation variants. Such evaluations can be represented as partially independent computational tasks, which makes them suitable for parallel execution using multifaceted process networks.

In the context of parallelizing dynamic sequential system programs, special importance is attached not only to improving execution performance but also to preserving the correctness of interaction between processes. Transforming a sequential program into a multifaceted process network involves separating functional fragments, organizing their concurrent execution, and ensuring data exchange between them. However, a larger number of simultaneously active processes complicates access to shared resources, message-passing channels, processing queues, and synchronization objects. Under such conditions, deadlock is not merely an external operating-system problem but an important safety constraint for the proposed parallel execution model.

Classically, a deadlock is a situation in which two or more processes hold already allocated resources and simultaneously wait for other resources held by processes from the same group. In a sequential program, such a situation may remain hidden because operations are executed one after another. After decomposition into parallel processes, however, some hidden dependencies may become resource interactions. Therefore, a parallelization method should consider not only information dependencies between instructions but also potential conflicts of resource access that may lead to cyclic waiting. In this work, this aspect is treated as a resource-safety extension of the parallelization procedure: the main method constructs the multifaceted process network, whereas the additional solver checks whether the obtained network can be executed safely under the available synchronization and resource constraints.

Thus, there is a need to improve the efficiency of program execution under conditions of widespread use of parallel computing architectures. The study of methods for parallelizing dynamic sequential system programs using multifaceted process networks makes it possible to develop approaches to transforming traditional software systems into more efficient parallel structures. This will contribute to improving program performance, increasing software scalability, and making fuller use of the computational capabilities of modern computer systems.

2. Related work

Program parallelization [18, 19, 25] is one of the key directions in the development of modern computing systems and software [4, 8, 12]. The rapid advancement of information technologies, the increasing volume of processed data, and growing performance requirements for software systems create a need for new approaches to organizing computational processes [10, 12, 26]. Modern computer systems widely employ multicore processors, graphics processing units, clusters, and distributed computing platforms that can perform a large number of operations simultaneously. However, the effective use of these hardware capabilities depends directly on whether software is able to support parallel computation.

Historically, a significant portion of software was designed for single-processor systems, where programs were executed sequentially. In such programs, instructions are performed one after another within a single control flow, and each subsequent operation can begin only after the previous one has been completed. This programming model is relatively simple to implement and analyze; however, it does not make full use of the potential of modern multicore computing systems. As a result, even when substantial computational resources are available, a program may use only a small part of them, leading to lower execution efficiency and increased data processing time [6, 11, 14].

For this reason, program parallelization is an important direction [1, 2, 5] in current computer science research. Parallelization refers to the process of transforming a sequential program or algorithm into a form in which separate parts of the computation can be executed simultaneously. The main objective of this process is to improve program execution performance through the simultaneous use of multiple computational resources. This can be achieved through multithreaded execution on a single processor, task distribution across several processor cores, or the use of distributed computing environments [3, 9].

The parallelization process [13, 21, 23] involves analyzing the structure of a program to identify independent or partially independent computational fragments. If individual operations or code blocks have no mutual dependencies, they can be executed in parallel without violating the logic of the program. Therefore, the task of parallelization is to identify such program sections and organize their parallel execution while taking possible constraints and dependencies into account [7, 15, 24].

The analysis of dependencies between operations [10, 17, 22] plays a particularly important role in this process. Various types of dependencies may exist in a program, including data, control, and resource dependencies. A data dependency occurs when one operation uses a result produced by another operation. In this case, the order of execution cannot be changed without affecting the correctness of the results. If operations are independent of each other, they can be executed simultaneously on different computational resources. Analyzing such dependencies is one of the key stages of program parallelization.

Another essential stage of program parallelization is the detailed analysis of dependencies between operations [13, 14]. These dependencies determine whether individual computational operations can be executed simultaneously or must be performed in a strictly defined sequence. If a dependency exists between operations, changing their execution order may lead to incorrect program results. Therefore, before parallel execution is organized, all possible dependencies in the program code must be carefully analyzed [16, 20].

In general, a dependency between operations arises when the execution of one operation affects the result of another. Such dependencies are most often associated with shared data or variables. If one operation changes the value of a variable and another operation uses that value, the execution order of these operations becomes important. In this case, parallel execution is not possible without additional coordination mechanisms.

In the theory of program parallelization [7, 10, 15], several main types of dependencies between operations are distinguished. The most important among them are data dependencies, control dependencies, and resource dependencies.

Within the analysis of parallelization approaches, resource dependencies should be considered separately from ordinary data dependencies. A data dependency describes the order of reading and writing variables, whereas a resource dependency reflects which files, memory areas, synchronization objects, channels, queues, or other system resources are used by a process. If such dependencies are ignored, formally independent fragments of a program may be mapped to parallel execution in a way that creates contention for resources and, in the worst case, forms the conditions for deadlock. Therefore, the detection and forecasting of resource conflicts should be included in the general procedure for constructing a multifaceted process network. In the proposed study, this requirement is incorporated not as an alternative to Bernstein's conditions but as their practical extension for system-level parallel execution: first, data dependencies determine which fragments may be executed concurrently; then, resource-state analysis verifies whether this concurrency is safe with respect to channels, locks, queues, and synchronization objects [27].

The analysis of existing approaches to program parallelization shows that further research in this area is necessary [1, 2, 5]. The development of effective methods for parallelizing dynamic sequential system programs is especially relevant, since such methods would make it possible to automate the detection of parallelism in program code and organize its efficient use [18, 19, 25]. Addressing this research problem requires the development of new program representation models, methods for analyzing dependencies between computations, and approaches to organizing interaction between parallel processes [3, 13, 24]. These issues form the basis of further research within this work and define its scientific focus.

3. Proposed approach

The development of new algorithmic approaches capable of automatically identifying sections of sequential program code that can be transformed into independent parallel execution threads remains a relevant research direction. The main idea is to create intelligent program analysis methods that can identify code fragments suitable for parallel execution without direct developer intervention and automatically transform them into structures optimized for multicore computing systems. In modern computing environments, where processors with multiple cores are widely used, such a modified program can be executed simultaneously on different computational resources, providing a significant increase in performance and hardware utilization efficiency. This study is based on the assumption that a significant part of sequential code may contain hidden parallelism that can be detected through formal analysis of dependencies between instructions and variables. In other words, if such parallelism is assumed to be present, its processing may enable program code to be executed in parallel.

The application of Bernstein’s conditions is particularly important in the process of automatic program parallelization performed by compilers or specialized optimization tools. During source code analysis, the software system determines the sets of read and write variables for each operator or instruction block and then checks whether the specified conditions are satisfied. If there are no conflicts in access to variables, the corresponding instructions can be placed in different execution threads. Bernstein’s conditions therefore define a formal criterion for the independence of operations in a program and serve as a basis for constructing dependency graphs that represent the structure of relationships between computational operations.

Based on Bernstein's conditions, generalized models of system program classes are formed for which the use of parallel computing is appropriate and effective. These classes include programs whose structure contains independent computational subtasks, absent or minimal dependencies between operations, and the possibility of dividing computations into a set of similar iterations or functional blocks. The three most characteristic program classes are programs with independent function calls, programs with independent instruction execution paths, and programs with iterative structures, or loops, in which each iteration can be executed independently of the others. For each of these classes, a formalized model is constructed to describe the computational structure, dependencies between variables, and parallelization conditions.

The first category of system programs includes programs in which a significant part of the computation is implemented as independent function calls. In such systems, the main algorithm can be represented as a composition of functions, each of which performs a separate part of the computation. If the results of these functions do not depend on one another or are used only after all function calls have been completed, their execution can be organized in parallel. This class of system programs can be defined by a set of functions as follows:

$$F=\left\{f_1,f_2,\dots,f_{n_F}\right\} \tag{1}$$

where each function f_i maps a set of input variables to a set of output variables: $f_i: X_i \rightarrow Y_i$; $X_i=\left\{x_{i,1},x_{i,2},\dots,x_{i,n_{X_i}}\right\}$; $Y_i=\left\{y_{i,1},y_{i,2},\dots,y_{i,n_{Y_i}}\right\}$; $i=1,2,\dots,n_F$; n_F – is the number of functions in the set of functions F; X_i - is the set of input variables of function f_i; and Y_i is the set of variables that form the result of the function execution.

Then, the general state of the program can be represented as a vector of variables as follows:

$$S=\left(s_1,s_2,\dots,s_{n_S}\right) \tag{2}$$

where s_j is the value of the j-th program variable.

After the execution of function f_i, the system state changes to a new state, which is defined as follows:

$$S'=f_i(S) \tag{3}$$

Parallel execution of functions f_i and f_j is possible only when the modified Bernstein conditions defined for the elements of the system program are satisfied:

$$\begin{gathered} R_i\cap W_j=\varnothing \\ R_j\cap W_i=\varnothing \\ W_i\cap W_j=\varnothing \end{gathered} \tag{4}$$

where Ri is the set of variables read by function f_i; Wi is the set of variables modified by function f_i; and Rj and Wj are the corresponding sets for function f_j.

The second category consists of system programs in which parallelism arises due to the presence of independent instruction execution paths within a single function or procedure. In this case, the algorithm is represented as a sequence of operators as follows:

$$P=\left(p_1,p_2,\dots,p_{n_P}\right) \tag{5}$$

where p_k is a separate program instruction, and n_P is the number of program instructions.

The program state after the execution of instruction p_k is defined by the transition function as follows:

$$s_{k+1} = p_k(s_k) \tag{6}$$

where s_k is the system state before the execution of instruction p_k, and s_{k+1} is the state after its execution.

The dependency between two instructions p_i and p_j is determined by the variables they use. Let $R(p_k)$ denote the set of variables read by instruction p_k, and let $W(p_k)$ denote the set of variables modified by this instruction. Instructions p_i and p_j can be executed in parallel if the following conditions are satisfied:

$$\begin{aligned} R(p_i) \cap W(p_j) &= \varnothing \\ R(p_j) \cap W(p_i) &= \varnothing \\ W(p_i) \cap W(p_j) &= \varnothing \end{aligned} \tag{7}$$

In this case, the set of instructions can be divided into independent subsets as follows:

$$P = P_1 \cup P_2 \cup \cdots \cup P_k \tag{8}$$

where each subset P_i is an independent execution path.

The maximum level of parallelism is determined by the number of such independent paths. An example of programs of this class is image processing, where different parts of an algorithm perform independent computations on different parameters. For example, during image frame processing, one program block may perform noise filtering, another may enhance contrast, and a third may compute the brightness histogram. If these operations do not modify shared variables, they can be executed in parallel.

The third category of system programs is defined by programs with iterative structures, where computations are performed in loops. In many algorithms, each loop iteration performs the same operations on different elements of a data set, which makes such algorithms natural candidates for parallelization. Formally, a loop can be defined as a sequence of iterations in which a function is executed at each iteration as follows:

$$\text{for } i = a, \ldots, b;\ S_{i+1} = F(S_i, i) \tag{9}$$

where i is the iteration index, a is the initial index value, b is the final value, S_i is the system state before the iteration, and F is the function that describes the computation in the loop body.

Let R_i be the set of variables read during iteration i, and let W_i be the set of variables modified during iteration i.

Iterations i and j can be executed in parallel if all three Bernstein conditions are satisfied. In addition, the values of the loop parameters must be determined before execution begins; that is, the following relation must hold:

$$N = b - a + 1 \tag{10}$$

where N is the number of loop iterations.

If these conditions are satisfied, the loop can be transformed into a set of independent tasks as follows:

$$T = \{T_1, T_2, \ldots, T_{n_T}\} \tag{11}$$

where each task T_i corresponds to the execution of one iteration.

An example of this class of programs is algorithms for processing arrays or matrices. For example, the computation of the sum of squares of array elements can be defined as follows:

$$S_{\sum MAS} = \sum_{i=1}^{n_{MAS}} m_i^2 \tag{12}$$

where m_i is an array element, n_{MAS} is the array size, and $S_{\sum MAS}$ is the result of computing the sum of squares of the array elements.

In this case, each operation $y_i = m_i^2$ can be executed in parallel, after which the results are combined using a reduction operation as follows:

$$S_{\sum MAS} = \sum_{i=1}^{n_{MAS}} y_i \tag{13}$$

The analysis of the structure of system programs makes it possible to distinguish three main models suitable for parallelization: the model of independent functional blocks, the model of independent instruction execution paths, and the model of independent loop iterations. Each of these models is characterized by its own formal conditions of dependencies between variables and operations, which makes it possible to determine the possibility of parallel execution and construct an optimal structure of a multithreaded algorithm. The use of such formalized models provides a basis for creating automated systems for program code analysis and optimization that can adapt traditional sequential algorithms to modern multicore computing systems.

In the classical formulation of Bernstein's conditions, three main criteria for operation independence are defined, making it possible to establish whether operations can be executed in parallel. These conditions are based on the analysis of the sets of variables read and modified by instructions and guarantee the absence of data access conflicts between parallel computations. However, in the practical application of methods for parallelizing system programs, another important aspect must be considered: the hardware limitations of the computing system, particularly the number of available processors or processor cores. Therefore, it is appropriate to supplement the classical conditions with a fourth condition related to the efficiency of executing parallel tasks with regard to hardware resources.

The fourth condition states that the number of parallel computational tasks formed as a result of program parallelization must be consistent with the number of available computational resources in the system. Let P denote the number of available processors or computational cores on which the program can be executed, and let T denote the number of independent tasks or threads obtained as a result of parallelization. The efficiency of parallel execution is then largely determined by the relationship between these values. If T significantly exceeds P, the system is forced to execute some tasks sequentially or perform frequent context switching between threads, which leads to additional execution management overhead. If T is significantly smaller than P, some computational resources remain unused, which also reduces the overall efficiency of program execution.

This condition can be expressed in the form of the following constraint:

$$T \leq P \cdot k \tag{14}$$

where T is the number of parallel tasks or threads; P is the number of available processors or cores; and k is a coefficient that defines the allowable excess of threads over processors and accounts for the characteristics of thread scheduling by the operating system.

In the simplest case, to achieve maximum utilization of computational resources, it is desirable for the number of threads to be close to the number of available cores, that is, for the following condition to hold:

$$T \approx P \tag{15}$$

In this case, each task can be executed on a separate core, providing the maximum level of parallelism without additional overhead related to thread scheduling.

If the total execution time of the program is considered, it can be represented as the sum of the execution times of the parallel and sequential parts of the algorithm. Let $T_s$denote the execution time of the sequential part of the program, T_p denote the execution time of the parallel part when using a single processor, and P denote the number of processors or cores. The theoretical execution time of the program under parallel execution can then be estimated as:

$$T_{exec} = T_s + \frac{T_p}{P} \tag{16}$$

This relation reflects a principle close to Amdahl's law, according to which the maximum gain from parallelization is limited by the fraction of sequential code. However, in real systems, this time also includes additional overhead caused by thread creation, synchronization, and data exchange between threads. Let these overhead costs be denoted by T_o. The actual execution time is then defined as:

$$T_{exec} = T_s + \frac{T_p}{P} + T_o \tag{17}$$

In this case, the fourth condition concerns not only the correspondence between the number of tasks and the number of processors, but also the requirement that the gain from parallel execution must exceed the overhead associated with organizing parallelism. That is, parallelization is appropriate only when the following condition is satisfied:

$$\frac{T_p}{P} + T_o < T_p \tag{18}$$

In other words, the total execution time of the parallel part of the program, including additional overhead, must be less than the time required to execute it sequentially.

The fourth condition therefore extends the classical Bernstein conditions by taking into account the hardware characteristics of the computing system and the actual overhead associated with organizing parallel execution. While the first three conditions determine the logical possibility of parallel instruction execution in terms of the absence of data dependencies, the fourth condition determines the practical feasibility of such parallelization with regard to available computational resources and the efficiency of processor core utilization. The combination of these four conditions makes it possible not only to identify independent computational structures in program code but also to ensure optimal use of the capabilities of modern multicore computing systems. This condition does not affect the number of defined classes; it only supplements the introduced classes with a new constraint.

The fifth constraint is Amdahl's law. Amdahl's law determines the potential speedup of an algorithm as the number of processors increases. Amdahl's law is one of the fundamental principles of parallel computing theory and defines the theoretical limit of program execution speedup when multiprocessor or multicore computing systems are used. It describes the relationship between the fraction of an algorithm that can be executed in parallel, the number of available processors, and the maximum possible performance gain. The main idea of this law is that, even with an unlimited number of processors, program performance is limited by the part of the algorithm that cannot be executed in parallel and must be performed sequentially.

Let T_1 denote the execution time of a given program on a single processor. This time can be represented as the sum of two components: the execution time of the sequential part of the algorithm and the execution time of the part of the program that can be parallelized. Let alpha denote the fraction of the sequential part of the program, that is, the part of the algorithm that cannot be executed in parallel. Accordingly, the fraction of the parallel part is $1-\alpha$. If P processors

or computational cores are used, the parallel part of the algorithm can theoretically be distributed among these processors. In this case, the execution time of the program on P processors can be defined as:

$$T_P = T_1 \cdot \left(\alpha + \frac{1-\alpha}{P}\right) \tag{19}$$

where T_p is the execution time of the program on P processors; T_1 is the execution time of the program on one processor; alpha is the fraction of the sequential part of the program; $1-\alpha$ is the fraction of the parallel part of the program; and P is the number of processors or cores.

One of the main characteristics of the efficiency of parallel computing is program execution speedup, which is defined as the ratio of the execution time on one processor to the execution time on P processors:

$$S(P) = \frac{T_1}{T_P} \tag{20}$$

By substituting the value of T_p into this formula, the classical expression of Amdahl's law is obtained:

$$S(P) = \frac{1}{\alpha + \frac{1-\alpha}{P}} \tag{21}$$

This formula shows that as the number of processors increases, program speedup also increases, but the rate of this increase gradually decreases. This is caused by the presence of the sequential part of the algorithm, which cannot be distributed among processors and is executed by only one thread.

If the limiting case is considered, where the number of processors tends to infinity, $P \to \infty$, the parallel part of the program is theoretically executed instantly, and the total execution time is determined only by the sequential part. In this case, the maximum possible speedup is equal to

$$S_{max} = \frac{1}{\alpha} \tag{22}$$

This means that even with an unlimited number of processors, program performance is limited by the size of the sequential part of the algorithm. For example, if 10% of the program is executed sequentially, $\alpha = 0.1$, then the maximum speedup will not exceed

$$S_{max} = \frac{1}{0.1} = 10 \tag{23}$$

regardless of how many processors are used.

The effective organization of parallel computation is based on combining structural analysis of program code, formal conditions for operation independence, and consideration of the hardware characteristics of the computing system. Three main models of system programs are identified for which parallelization is the most appropriate and technically feasible. Each of these models is characterized by specific features of algorithm structure and by the way computational processes are organized, which determines the potential for executing separate parts of a program in parallel.

The first model covers system programs built on independent functional modules or function calls. In such programs, the main algorithm can be represented as a set of functional blocks, each of which performs a separate computational task and operates on its own subset of data. If there are no data dependencies between these functional blocks, the blocks can be executed simultaneously in different threads or on different computational cores. This model is typical of data processing

systems, analytical programs, statistical computation programs, and other systems in which the results of individual computations can be obtained independently of one another.

The second model of system programs is based on the presence of independent instruction execution paths within a single algorithm or procedure. In this case, a program can be considered as a sequence of operators between which data dependencies may or may not exist. If separate groups of instructions do not use shared variables or modify each other's execution results, these instructions can be executed in parallel. The analysis of such structures makes it possible to identify independent subgraphs in the program dependency graph, enabling simultaneous execution of different parts of the algorithm. This model is characteristic of signal processing systems, computational modules in complex software systems, and programs in which separate computational stages do not depend on the results of other stages.

The third model of system programs is associated with iterative structures, particularly loops, in which the same operations are performed on different data elements. In such cases, each loop iteration can be considered as a separate computational task. If there are no data dependencies between iterations, they can be executed in parallel on different computational cores. This model is typical of array processing algorithms, matrix computations, physical process modeling, image processing, and other tasks involving large volumes of similar computations. This model is most commonly used in high-performance computing systems because it enables efficient program scaling as the number of processors increases.

Effective parallelization of system programs is therefore based on an integrated approach that combines structural analysis of algorithms, formal criteria for computational independence, and consideration of the hardware constraints of computing systems. The identification of three main program models, the application of Bernstein's conditions to analyze dependencies between operations, and the consideration of the number of computational cores and the limitations defined by Amdahl's law make it possible to form a comprehensive methodology for program analysis and optimization in multicore computing environments. The implementation of this approach creates prerequisites for improving the performance of software systems, using hardware resources more efficiently, and adapting software to modern parallel computing architectures.

After the construction of the multifaceted process network, the method is supplemented with an auxiliary deadlock-aware verification stage. This stage is not a separate parallelization mechanism and does not replace dependency-graph construction. Its role is to evaluate whether the already formed set of interacting processes can be safely executed in parallel under the selected resource allocation and synchronization scheme. Therefore, the solver is placed between decomposition into multifaceted processes and the final mapping of those processes onto computational cores or nodes.

The input of the solver is a set of process signatures generated from the multifaceted process network. A process signature combines the computational role of a process with its resource-interaction characteristics: the occupied and requested resources, the intensity of message exchange, the frequency of accesses to channels, the state of processing queues, the type of synchronization, and dependencies on results produced by other processes. In this form, the signature links classical program-dependency analysis with runtime resource conditions that may cause cyclic waiting.

Since the states of dynamic system programs are often uncertain and not strictly separated, the solver uses intelligent analysis of resource interaction. Accumulated or simulated execution examples are clustered by Fuzzy C-Means to identify typical behavior scenarios: normal execution, increased resource contention, potentially unsafe waiting, and states close to deadlock. The obtained clusters serve as a basis for forming a fuzzy rule base, which allows the solver to evaluate gradual transitions between safe and unsafe resource states instead of using only rigid threshold rules.

The output of the solver is a deadlock-risk estimate for individual processes or for the process network as a whole. A low-risk result allows the formed network to be mapped onto processor cores or computational nodes. If the risk exceeds an acceptable threshold, the method does not

reject the parallelization result entirely; instead, it introduces corrective actions, such as changing the process start order, limiting simultaneous access to selected resources, adding synchronization mechanisms, serializing unsafe fragments, or redistributing processes between computational resources.

The diagram in Figure 1 shows an example of a multifaceted process network used to organize parallel computation. Each hexagonal block in the diagram represents a separate multifaceted process that performs computation over a specific part of the algorithm's iteration space. The central process may act as a coordinating node or as a process that handles the main part of the data. The arrows between the blocks represent data transfer channels between processes. Through these channels, the processes exchange intermediate computation results. The direction of the arrows indicates the direction of information transfer from one process to another. This interaction ensures the correct order of operation execution according to data dependencies.

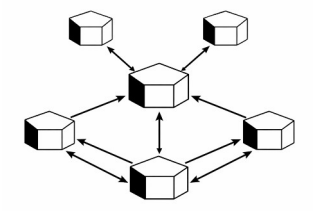

Figure 1: Example of a multifaceted process network.

External processes can perform computations in parallel by processing different parts of the task. After the computations are completed, the results are transferred to other processes, which use them to perform the subsequent stages of the algorithm. This makes it possible to significantly reduce the overall program execution time. The diagram therefore demonstrates the principle of organizing a parallel computing system in which several multifaceted processes operate simultaneously and interact through a data transfer network. This structure ensures efficient use of the computational resources of multiprocessor systems.

The method for parallelizing dynamic sequential system programs using multifaceted process networks can improve the efficiency of modern software systems. Its application enables a transition from the traditional sequential execution model to a flexible parallel computing architecture that corresponds to current trends in the development of modern computer systems and provides more complete use of their computational capabilities.

Steps:

1. Classification of the system program. The first stage of the method is to determine the class of the system program according to the generalized model of system programs, which divides software into three main classes depending on the nature of the functions performed. The result of this step is the formation of an initial program model, which makes it possible to define the general strategy for further parallelization and select appropriate methods for analyzing its structure.
2. Analysis of the sequential program structure. The second step of the method is a detailed analysis of the sequential program structure in order to identify internal dependencies between operations and data flows. At this stage, the source code, the execution logic of the algorithm, and the relationships between separate parts of the program are examined. As a result of this step, an understanding of the program's computational structure is formed, and key sections where parallelization is potentially possible are identified.
3. Construction of a computational dependency graph. At this stage, the results of the previous analysis are formalized as a computational dependency graph. This graph is a mathematical model of the program execution structure, in which vertices correspond to individual operations or functional blocks, while edges represent dependencies between them. As a result, a structured program model is formed that represents all necessary

relationships between its computational elements and serves as a basis for further decomposition of the program into separate processes.

4. Decomposition of the program into functional blocks. After the dependency graph has been constructed, the program is decomposed into functional blocks that can be implemented as separate computational processes. Decomposition involves dividing the program into relatively independent fragments, each of which performs a certain logically complete part of the algorithm. Based on the decomposition, a set of functional blocks is formed as candidates for transformation into parallel processes.
5. Formation of multifaceted processes. The next step is to transform the program's functional blocks into multifaceted processes. In the proposed approach, a multifaceted process is considered as a universal computational element that combines several aspects of operation. As a result of this step, a set of interacting processes is formed, representing the main computational components of the future parallel system.
6. Construction of a multifaceted process network. After the individual processes have been formed, a network of their interaction is constructed. A multifaceted process network is a structured model of parallel computation in which processes act as nodes, while the connections between them are implemented as data transfer channels or synchronization mechanisms. As a result, a flexible computational structure is formed that enables effective use of the capabilities of parallel computing systems.
7. Identification of parallel execution regions and resource-safety verification. After the process network has been constructed, the computational sections that can be executed in parallel are identified. For this purpose, the network structure and the dependencies between processes are analyzed. At the same stage, generated process signatures are submitted to the deadlock-aware solver. The solver estimates whether the planned interaction between processes may create a cyclic resource-waiting situation. The result of this step is therefore twofold: the set of parallel execution regions and, if necessary, corrective actions that make the selected parallel structure safe before it is mapped onto computational resources.
8. Mapping the process network onto computational resources. The final step of the method is to map the formed multifaceted process network onto the actual computational resources of the system. At this stage, it is determined which processes will be executed on specific processor cores or computational nodes. The result of this step is a parallel implementation of the program adapted to a specific computing system and capable of efficiently using its resources.

In cybersecurity-oriented applications, this decomposition can support the parallel evaluation of alternative decoy-network configurations, where separate processes analyze different attacker paths, defensive responses, or resource allocation strategies.

The method for parallelizing dynamic sequential system programs is based on representing the program execution structure as a multifaceted process network. This approach makes it possible to formalize the complex behavior of system programs characterized by conditional branches, dynamic data structures, and variable execution flows. Unlike traditional parallelization methods, which are mainly oriented toward static algorithms, the proposed method takes into account the dynamic nature of system software and enables the computational structure to be adapted during program execution.

The method involves the sequential execution of several stages, beginning with the classification of the system program according to its functional purpose and the analysis of the sequential algorithm structure. The subsequent construction of a computational dependency graph makes it possible to formalize the relationships between program operations and identify potential opportunities for parallel execution. Based on this graph, the program is decomposed into functional blocks that are transformed into multifaceted processes. Each of these processes

combines computational, communication, and control aspects of operation, ensuring the flexibility and scalability of the computational structure.

The application of the proposed method therefore makes it possible to transform dynamic sequential system programs into parallel computational structures without significantly changing their operational logic. The use of multifaceted process networks enables formal program analysis, increases the level of execution parallelism, and contributes to more efficient use of the hardware resources of modern computer systems. This creates prerequisites for improving software performance and scalability under the development of multicore and distributed computing platforms.

4. Performance evaluation and experiment

The experimental part of the implementation of the multifaceted process network software system was conducted to evaluate the efficiency of parallel execution of dynamic sequential system programs and to verify the correctness of data processing in a stream-based system. The main task was to model a message stream simulating system logs and to determine the performance of the multifaceted process network when processing this stream.

During the experiment, the software system implemented in C++ created a network of processes that performed separate functions: message generation (Producer), message parsing (ParserProcess), filtering (FilterProcess), analysis (AnalyzerProcess), statistics collection (StatisticsProcess), result aggregation (AggregatorProcess), and logging of processed messages (LoggerProcess). All processes were executed in parallel as threads and exchanged messages through thread-safe channels. The use of channels provided synchronization, prevented data access conflicts, and supported message buffering.

To evaluate performance, the experiment was conducted with different input data volumes: 100, 200, and 500 messages. The processing time was measured from the start of message generation to the completion of output by LoggerProcess. In addition, the correctness of the computations was verified: messages passed correctly through all processes, filters excluded data according to the specified conditions, and aggregate indicators matched the expected values.

The experimental results showed that parallel execution significantly reduces processing time compared with sequential execution. An increase in the number of messages leads to a proportional increase in processing time, but the speedup achieved through parallelism remains significant. It was found that, with a large volume of messages, data transfer channels may become a bottleneck, which emphasizes the need to optimize queues and buffering.

The experimental results are presented in tables. The first table, Table 1, shows the processing time for different message volumes, while the second table, Table 2, presents the estimated speedup compared with sequential execution.

Table 1
Message processing time in a multifaceted process network

Number of messages	Processing time for parallel execution (s)	Processing time for sequential execution (s)
100	2.1	5.6
200	4.3	11.2
500	10.8	28.0

The experiment confirmed that the implemented system provides efficient parallel execution, maintains the correctness of data processing, and demonstrates significant speedup compared with

the sequential approach. These results indicate the practical suitability of the proposed method for optimizing computations in system software complexes and for creating scalable software platforms for processing large data streams.

Table 2
Message processing time in a multifaceted process network

Number of messages	Speedup (sequential / parallel)
100	2.1
200	4.3
500	10.8

To evaluate the efficiency of parallel program execution, standard metrics from the theory of parallel computing are used:

1. Sequential program execution time T_s, which is the time required to process all tasks on a single processor.
2. Parallel program execution time T_p, which is the time required for the multifaceted process network to process the same tasks using n threads or cores.

Speedup S is the ratio of sequential execution time to parallel execution time and is defined as follows:

$$S = \frac{T_s}{T_p} \tag{24}$$

Efficiency E is the ratio of speedup to the number of parallel threads and is defined as follows:

$$E = \frac{S}{n} = \frac{T_s}{n \cdot T_p} \tag{25}$$

The resource utilization coefficient U reflects processor load and the coordination of data transfer channels. It is defined as the ratio of the time during which processors perform computations to the total processing time:

$$U = \frac{T_o}{T_p} \cdot 100\% \tag{26}$$

The summary tables of the results, Tables 3 and 4, take into account the number of threads/cores and present the processing time, speedup, and efficiency of parallel execution.

The experimental results show that the multifaceted process network provides a significant speedup in message stream processing compared with sequential execution. The greatest performance increase is observed for a small number of messages, while efficiency slightly decreases as the data volume grows due to the overhead associated with synchronization and message transfer between threads. The efficiency graphs are shown in Figure 2 and Figure 3.

Table 3
Processing Time and Speedup

Number of messages	Threads/cores	$T_s(s)$	$T_p(s)$	Speedup S
100	4	5.6	2.1	2.7
200	4	11.2	4.3	2.6
500	4	28.0	10.8	2.6

Table 4
Parallel Execution Efficiency

Number of messages	Threads/cores	Speedup S	Efficiency $E(\%)$
100	4	2.7	67.5
200	4	2.6	65.0
500	4	2.6	65.0

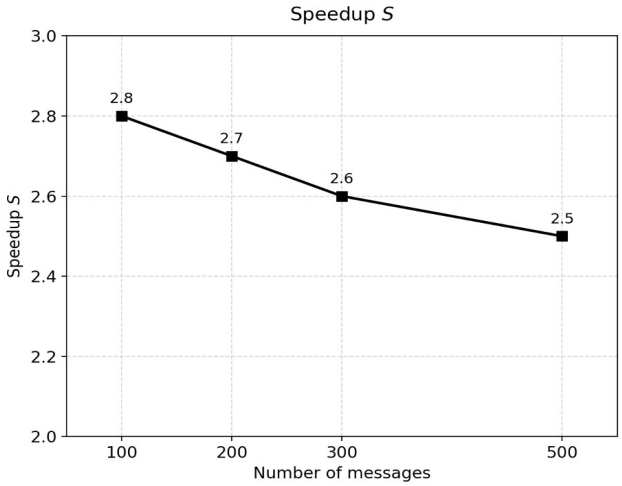

Figure 2: Speedup depending on the number of messages.

The average program speedup S is approximately 2.6–2.7 when four threads are used, which confirms the correctness of the multifaceted process network structure. The efficiency E remains at the level of 65–67%, indicating effective use of hardware resources and moderate time overhead for managing threads and data transfer channels.

In the proposed system, the term “message” denotes a data unit transferred between processes in a multifaceted process network. Each message contains certain information that must be processed, such as a numerical value, a textual description of an event, or other system log data. In the C++ implementation, each message is represented by an object of the Message class with the fields id, payload, and value. The system processes receive messages through Channel channels, perform computations or transformations, and then pass each message to the next process in the network.

Therefore, a message is a key structure for organizing process interaction and supporting parallel processing of data streams.

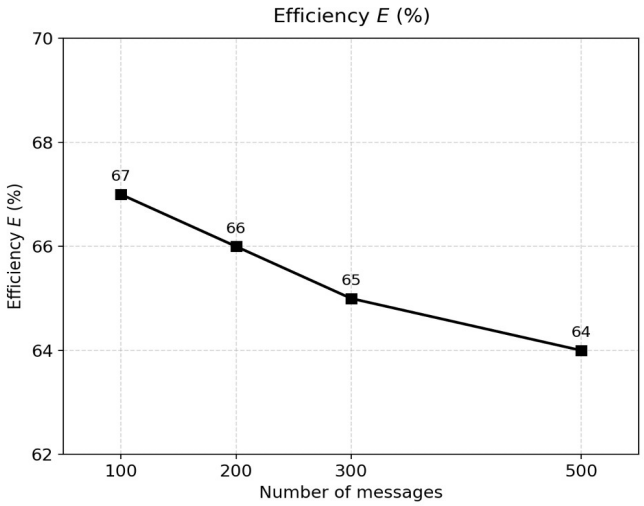

Figure 3: Efficiency depending on the number of messages.

The decrease in system efficiency as the number of messages increases is explained by the growth of overhead associated with thread management, synchronization, and channel queues as the load increases. Each process, operating in its own thread, must interact with channel queues to receive and transmit messages. With a small number of messages, synchronization overhead is minimal, and the processes spend almost all their time performing computations. However, when the number of messages increases significantly, channel queues become filled, and processes begin to wait either until space becomes available for new messages or until other processes read messages from the queue. As a result, part of the processor time is spent not on computation but on waiting for access to resources, which is reflected in the decrease in overall efficiency.

In addition, with a large message stream, extra overhead arises from copying or moving messages between channels and from managing the lifecycle of Message objects, which further increases overhead and reduces the efficiency of hardware resource utilization. Another reason for the decrease in efficiency is thread contention for access to shared resources, such as channel queues or synchronization mechanisms (mutex, condition_variable). When the number of messages is large, several threads simultaneously attempt to read or write messages, which creates delays due to resource locking. Although these locks are short-lived, their accumulated effect noticeably affects overall system performance. In other words, even if the processors are not idle, the overhead associated with synchronization and channel processing reduces the efficiency of parallel execution. Moreover, when large data volumes are processed, channels become a bottleneck because the message processing rate of a particular process may lag behind the rate of generating new messages, leading to temporary queue accumulation and increased waiting time. Another aspect that affects efficiency is the structure of the messages themselves. In the implementation, messages contain both numerical and textual data. With a large message stream, processes spend a considerable amount of time copying text strings and managing memory, which also increases additional overhead. As the message volume increases, these costs grow nonlinearly, and part of the processor time is spent on auxiliary operations instead of the computations that are the main task of the processes. Therefore, even under parallel execution, efficiency does not increase proportionally with the number of messages; instead, it slightly decreases due to overhead and resource contention.

Since the proposed method includes a resource-safety verification stage, the experimental interpretation should not be limited to speedup and efficiency only. The behavior of channel queues, short-term blocking on mutexes, and waiting for message transfer can also be interpreted

as input characteristics for the deadlock-aware solver. In the presented implementation, these parameters are consistent with the process-signature model because they describe not only the computational workload of each process but also its interaction with shared resources.

An additional verification scenario can therefore be formed on the basis of the same process network by comparing normal message processing, increased queue contention, and artificially created cyclic waiting between processes. In such a scenario, the solver does not change the measured processing time directly; rather, it acts as a preventive control stage. Its task is to indicate whether the planned parallel execution remains safe and, when a high-risk state is detected, to recommend corrective measures before the network is executed on the available cores. This makes the deadlock-aware component consistent with the experimental model without shifting the main focus of the experiment away from parallelization performance.

The method for parallelizing dynamic sequential system programs through a multifaceted process network is therefore effective and makes it possible to improve the performance of system programs without losing data processing correctness.

The experimental verification of the multifaceted process network implementation confirmed the effectiveness of the proposed method for parallelizing dynamic sequential system programs. During the experiments, it was established that parallel execution of processes within the network significantly reduces message processing time compared with sequential execution. Even when the message volume increased to 500 units, a speedup of approximately 2.6–2.7 times was observed, confirming the performance of the system and the correctness of computation distribution among threads.

The efficiency analysis showed that, with a small number of messages, hardware resources are used to the greatest extent, while synchronization and channel processing overhead remains minimal. As the load increases, part of the execution time is spent waiting for access to channel queues, managing threads, and buffering data, which slightly reduces efficiency. Nevertheless, even with these overhead costs, the system maintains a high level of resource utilization and demonstrates stable speedup.

5. Conclusions

The features of dynamic sequential system programs were examined, and the challenges associated with their effective parallelization were identified. It was found that the main issues hindering effective parallelization include complex data dependencies, an irregular computational structure, dynamic task creation, the need to synchronize access to shared resources, and the risk of unsafe resource interaction between concurrently executed processes. These factors complicate automatic parallelization and necessitate specialized approaches for organizing parallel execution.

A method for parallelizing dynamic sequential system programs using multifaceted process networks has been developed. This approach allows for the formalization of the computational structure, identification of independent or partially independent program fragments, and organization of their execution as interacting processes that run in parallel.

A model has been established to represent a sequential system program as a network of interacting processes, with specific mechanisms for their interaction and synchronization defined. This model ensures correct data exchange, execution coordination, and prevention of resource access conflicts, thereby enhancing program scalability and effectively utilizing the computational resources of multicore systems. The model is additionally supported by a deadlock-aware verification stage that checks the safety of the formed process network before final resource mapping.

The effectiveness of the proposed method for improving program execution performance on multicore computing systems has been investigated. Evaluations showed that the use of multifaceted process networks significantly enhances program execution performance on multicore systems. The results confirm the viability of this approach for optimizing the execution of complex dynamic system programs and improving the efficiency of hardware resource utilization.

Further research will focus on the parallelization of processes in distributed environments, on the extended empirical validation of the deadlock-aware solver, and on investigating the applicability of the proposed model to cybersecurity tasks, particularly the synthesis and evaluation of honeypot and decoy-network configurations based on attacker-defender interaction models.

As part of the developed method, the deadlock-aware solver is considered a regular auxiliary component of the multifaceted process network construction procedure. It allows the method to analyze not only information dependencies between program fragments but also resource interactions that arise during parallel execution. The use of process signatures, fuzzy clustering, and a fuzzy rule base creates a basis for early detection of unsafe resource situations and for correcting the parallel structure before a deadlock actually occurs. Thus, the solver complements the main performance-oriented objective of parallelization with a resource-safety criterion.

Declaration on Generative AI

During the preparation of this work, the author(s) used Grammarly in order to proofread and ensure the correctness of the English language. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] Y. An, X. Chen, K. Gao, Y. Li, L. Zhang, Multiobjective flexible job-shop rescheduling with new job insertion and machine preventive maintenance, IEEE Trans. Cybern. 53 (2023) 3101–3113. doi:10.1109/TCYB.2022.3151855.

[2] Y. An, X. Chen, Y. Li, Y. Han, J. Zhang, H. Shi, An improved non-dominated sorting biogeography-based optimization algorithm for (hybrid) multi-objective flexible job-shop scheduling problem, Appl. Soft Comput. 99 (2021) 106869. doi:10.1016/j.asoc.2020.106869.

[3] M. Bi, I. Kovalenko, D. M. Tilbury, K. Barton, Dynamic distributed decision-making for resilient resource reallocation in disrupted manufacturing systems, Int. J. Prod. Res. 62 (2024) 1737–1757. doi:10.1080/00207543.2023.2200567.

[4] I. Ramskyi, A. Drozd, O. Lyhun, O. Ponochovna, System for cybersecurity evaluation of corporate networks, Computer Systems and Information Technologies 2 (2025) 123–131. doi:10.31891/csit-2025-2-14.

[5] Y. Li, W. Huang, R. Wu, K. Guo, An improved artificial bee colony algorithm for solving multi-objective low-carbon flexible job shop scheduling problem, Appl. Soft Comput. 95 (2020) 106544. doi:10.1016/j.asoc.2020.106544.

[6] R. Chen, T. C. E. Cheng, C. T. Ng, J.-Q. Wang, H. Wei, J. Yuan, Rescheduling to trade off between global disruption of original jobs with flexibility and scheduling cost of new jobs, Omega 128 (2024) 103114. doi:10.1016/j.omega.2024.103114.

[7] X.-L. Chen, J.-Q. Li, Y. Xu, Q-learning based multi-objective immune algorithm for fuzzy flexible job shop scheduling problem considering dynamic disruptions, Swarm Evolut. Comput. 83 (2023) 101414. doi:10.1016/j.swevo.2023.101414.

[8] O. Savenko, A. Nicheporuk, I. Hurman, S. Lysenko, Dynamic signature-based malware detection technique based on API call tracing, CEUR-WS 2393 (2019) 633–643.

[9] T. Qin, R. Du, A. Kusiak, H. Tao, Y. Zhong, Designing a resilient production system with reconfigurable machines and movable buffers, Int. J. Prod. Res. 60 (2022) 5277–5292. doi:10.1080/00207543.2021.1953715.

[10] A. Esteso, D. Peidro, J. Mula, M. Díaz-Madroñero, Reinforcement learning applied to production planning and control, Int. J. Prod. Res. 61 (2023) 5772–5789. doi:10.1080/00207543.2022.2104180.

[11] J. M. Framinan, V. Fernandez-Viagas, P. Perez-Gonzalez, Using real-time information to reschedule jobs in a flowshop with variable processing times, Comput. Ind. Eng. 129 (2019) 113–125. doi:10.1016/j.cie.2019.01.036.

[12] G. Markowsky, O. Savenko, S. Lysenko, A. Nicheporuk, The technique for metamorphic viruses' detection based on its obfuscation features analysis, CEUR-WS 2104 (2018) 680–687.
[13] Y. Fang, C. Peng, P. Lou, Z. Zhou, J. Hu, J. Yan, Digital-twin-based job shop scheduling toward smart manufacturing, IEEE Trans. Ind. Inform. 15 (2019) 6425–6435. doi:10.1109/TII.2019.2938572.
[14] M. Ghaleb, H. Zolfagharinia, S. Taghipour, Real-time production scheduling in the industry 4.0 context: addressing uncertainties in job arrivals and machine breakdowns, Comput. Oper. Res. 123 (2020) 105031. doi:10.1016/j.cor.2020.105031.
[15] J.-P. Huang, L. Gao, X.-Y. Li, C.-J. Zhang, A cooperative hierarchical deep reinforcement learning based multi-agent method for distributed job shop scheduling problem with random job arrivals, Comput. Ind. Eng. 185 (2023) 109650. doi:10.1016/j.cie.2023.109650.
[16] O. Savenko, S. Lysenko, A. Nicheporuk, B. Savenko, Approach for the Unknown Metamorphic Virus Detection, Proceedings of the 8-th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (2017) 71–76. doi:10.1109/IDAACS.2017.8095052
[17] S. Hwangbo, J. J. Liu, J.-H. Ryu, H. J. Lee, J. Na, Production rescheduling via explorative reinforcement learning while considering nervousness, Comput. Chem. Eng. 186 (2024) 108700. doi:10.1016/j.compchemeng.2024.108700.
[18] C. D. Hubbs, C. Li, N. V. Sahinidis, I. E. Grossmann, J. M. Wassick, A deep reinforcement learning approach for chemical production scheduling, Comput. Chem. Eng. 141 (2020) 106982. doi:10.1016/j.compchemeng.2020.106982.
[19] B. A. Han, J. J. Yang, A deep reinforcement learning based solution for flexible job shop scheduling problem, Int. J. Simul. Model. 20 (2021) 375–386. doi:10.2507/IJSIMM20-2-CO7.
[20] S. Lysenko, K. Bobrovnikova, R. Shchuka, O. Savenko, A Cyberattacks Detection Technique Based on Evolutionary Algorithms, 11th International Conference on Dependable Systems, Services and Technologies (DESSERT) 1 (2020) 127–132. doi:10.1109/DESSERT50317.2020.9125016
[21] S. Jun, S. Lee, H. Chun, Learning dispatching rules using random forest in flexible job shop scheduling problems, Int. J. Prod. Res. 57 (2019) 3290–3310. doi:10.1080/00207543.2019.1581954.
[22] Y. Li, S. Carabelli, E. Fadda, D. Manerba, R. Tadei, O. Terzo, Machine learning and optimization for production rescheduling in industry 4.0, Int. J. Adv. Manuf. Technol. 110 (2020) 2445–2463. doi:10.1007/s00170-020-05850-5.
[23] K. Li, Q. Deng, L. Zhang, Q. Fan, G. Gong, S. Ding, An effective MCTS-based algorithm for minimizing makespan in dynamic flexible job shop scheduling problem, Comput. Ind. Eng. 155 (2021) 107211. doi:10.1016/j.cie.2021.107211.
[24] K. Peng, Q.-K. Pan, L. Gao, X. Li, S. Das, B. Zhang, A multi-start variable neighbourhood descent algorithm for hybrid flowshop rescheduling, Swarm Evolut. Comput. 45 (2019) 92–112. doi:10.1016/j.swevo.2019.01.002.
[25] K. Lei, P. Guo, W. Zhao, Y. Wang, L. Qian, X. Meng, L. Tang, A multi-action deep reinforcement learning framework for flexible job-shop scheduling problem, Expert Syst. Appl. 205 (2022) 117796. doi:10.1016/j.eswa.2022.117796.
[26] O. Bokhonko, O. Atamaniuk. Method for synthesis of a scalable architecture of a distributed cs, resistant to social engineering attacks. Computer Systems and Information Technologies 4 (2025) 60–76. https://doi.org/10.31891/csit-2025-4-7.
[27] O. Savenko, S. Lysenko, A. Nicheporuk, B. Savenko, Approach for the unknown metamorphic virus detection. In: Proceedings of the 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS 2017), Bucharest, Romania, September 21–23, 2017, pp. 71–76.