UDC 004.42 https://doi.org/10.20535/2786-8729.6.2025.333586

ENVIRONMENT FOR TUNING PARAMETERS OF A MULTITHREADED PROGRAM DEVELOPED USING A DEPENDENCY GRAPH

Kostiantyn Nesterenko *
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
https://orcid.org/0000-0003-3921-4324

Inna V. Stetsenko
National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
https://orcid.org/0000-0002-4601-0058

*Corresponding author: k.nesterenko@kpi.ua

With the advent of multi-core central processors, multithreading has become the most widespread practice for improving program execution performance. However, the development of a multithreaded program remains a rather complex process. To simplify this process and enhance the performance of the resulting program, various methods for managing thread-based execution are often employed.

One such method is the method of managing the execution of tasks of a multithreaded program according to a given dependency graph. This method significantly reduces the resource intensity of program development and increases program performance by employing a lockless approach to multithreaded programming.

Nevertheless, the challenge of efficient utilization of computational resources remains relevant and can only be addressed through the careful design of parallel computations. In particular, identifying the configuration parameters for a multithreaded program that ensure optimal resource utilization is a resource-intensive and complex task, even for highly qualified specialists.

This study examines existing approaches to tuning the parameters of multithreaded programs to achieve the most efficient execution. It proposes the use of an environment for tuning multithreaded program parameters based on the method of managing the execution of tasks of a multithreaded program according to a given dependency graph. The accuracy of the resource efficiency metrics obtained through this environment was experimentally validated. A practical example demonstrates the application of the environment in the development of a multithreaded program. The use of the environment also facilitates the configuration process of multithreaded program parameters.

Keywords: software, multithreading, computational resources, dependency graph, C++ parallel programming.

1. Introduction

The development of a multithreaded program is a complicated process, even for an experienced specialist. To simplify and enhance this process, various methods for managing program execution in threads are employed. One such prominent method is the method of managing the execution of tasks of a multithreaded program according to a given dependency graph [1]. This method allows presenting an execution flow of a multithreaded program as a directional acyclic graph (DAG) and performing tasks according to the given DAG. Application of this method significantly simplifies the development process, as well as increasing overall program performance by utilizing the lockless approach to multithreaded execution.

However, the task of determining the optimal amount of computational resources required for program execution remains and often presents a non-trivial and resource-consuming challenge [2].

For any given codebase, gradually increasing the number of threads eventually leads to diminishing returns in performance gains [3]. It is important to recognize that a higher thread count incurs greater overhead associated with deployment and maintenance of the software solution. Moreover, increasing the number of threads also raises the hardware requirements (and thus the cost) of the system on which the program runs [4]. Therefore, identifying a configuration that ensures both satisfactory performance, defined by the program’s non-functional requirements, and efficient utilization of computational resources remains a highly relevant task.

In addressing this challenge, two general approaches can be distinguished: first, the application of computational modeling methods during the software design phase; and second, experimental or automated tuning of parameters in an already developed software system [5]. However, both approaches suffer from a common drawback – significant resource consumption.

Thus, reducing the resource consumption of the process of tuning the parameters of a multithreaded program remains a relevant scientific problem.

2. Literature review and problem statement

2.1. Methods for modeling software computations

One of the most widely used and effective methods for system modeling is the application of Petri nets. In particular, stochastic Petri nets allow sufficiently accurate modeling of asynchronous and parallel systems, as they account for computational execution delays and the stochastic nature of resource acquisition [6]. The number of tokens in specific places of the net's initial state enables the representation of available resources, such as logical cores of central processing unit (CPU), and allows tracking their occupancy during computation.

However, this approach also has limitations. Chief among them is the high qualification requirement for specialists, including a solid theoretical background in automata theory and formal methods in software engineering. Furthermore, although software tools for designing and simulating Petri nets exist, such as CPN IDE (Colored Petri nets Integrated Development Environment) [7], modeling parallel computations according to the program code still demands substantial expert effort. For this reason, such solutions are unsuitable for small companies or startups, where rapid software development is a critical priority.

Another drawback of modeling computations using timed Petri nets lies in the definition of delay parameters. The accuracy of such models directly depends on how closely the defined delays reflect the actual execution time of specific functions or processes within the program. These delays are also influenced by the characteristics of the hardware on which the software is executed, making the development of a precise computational model particularly challenging during the prototyping phase or when designing new software.

2.2. Software tools for program profiling

Currently, a wide range of tools are available to developers for debugging and performance tuning. Some are built into integrated development environments (e.g., Performance Profiler in Visual Studio [8]) or game engines (e.g., Unreal Insights for Unreal Engine 5 [9]), while others exist as independent solutions that can be integrated into software products (e.g., RAD Telemetry Performance Visualization System [10]). In most cases, these tools record information during program execution about when and in which thread a specific call occurred, as well as how long it took to execute. Developers can then analyze this data to identify performance bottlenecks [11].

Although this approach ensures accurate insight into program execution behavior, it has a notable drawback. When system parameters change – for instance, when the number of available threads is increased or decreased – performance analysis must be repeated from scratch. This involves re-executing the program and reanalyzing the data to assess the impact of these changes on performance. The process itself is resource-intensive, and its efficiency depends heavily on the experience and qualifications of the specialist conducting the analysis [13]. The presence of parallel computations in the program adds further complexity due to the inherent stochasticity in how different program segments are executed.

Thus, based on the analysis of existing methods for assessing program performance and resource utilization efficiency, it can be concluded that their primary drawback lies in their high resource demands and the significant level of expertise required for their effective use.

Accordingly, the problem of reducing the resource intensity of configuring parameters in multithreaded programs remains an important direction for further research.

3. The aim and objectives of the study

The aim of this research is to reduce the resource intensity of the process of configuring parameters for multithreaded programs developed using a dependency graph.

To achieve the goal, the following tasks are set:

– to create an environment for tuning parameters of a multithreaded program that allows replaying the execution of a multithreaded program, changing its parameters, and running simulations with modified parameters,

– to conduct an experimental investigation of the accuracy of results provided by the environment.

4. The study materials and methods of developing the environment for tuning parameters of the multithreaded program

4.1. General overview of the environment

The design of the environment is based on a task execution control method for multithreaded programs using a predefined dependency graph [1]. This method represents the set of subtasks in a multithreaded program as a directed acyclic graph, where the graph's nodes correspond to individual subtasks and the edges denote dependencies between them.

The proposed approach integrates data collection (see Section 4.3) on execution time and computational resource utilization. Based on the collected data, the environment is capable of reproducing the program execution process, enabling developers to visually identify performance bottlenecks. Additionally, the environment provides key statistical insights, including total execution time, the working and idle durations of each thread, and an estimate of overall resource utilization efficiency.

Moreover, with empirical data on the execution time of each subtask, the environment enables simulation of program execution under varying system resource configurations. This allows the developer to alter the number of available threads for executing the task graph and observe simulation results under different system configurations – without needing to run the actual program.

4.2. Simulation Algorithm Development

As mentioned earlier, the simulation algorithm is built upon the task execution control method using a predefined dependency graph [1]. To ensure high accuracy in simulation results, the algorithm mirrors the logic of the original method.

The simulation algorithm implements discrete-event simulation. Simulation time progresses between events corresponding to the completion of a subtask by one of the threads.

To simulate the behavior of individual threads, the '`SimulationWorker`' class is implemented. This class emulates the processing logic of a node and determines the expected completion time for that node. For representing the state of nodes in the dependency graph, the '`GraphNode`' class is developed. Instances of this class maintain the state of a node as well as its dependency information. A node can be in '`blocked`', '`partially unblocked`', '`ready`', '`executing`', or '`processed`' state. The state of a node is updated via the '`updateSimulation`' method, which receives the current simulation time as input.

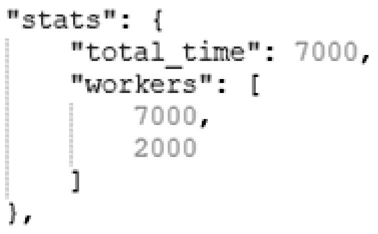
After each state update, the next simulation event time is calculated as the nearest scheduled task completion among all active threads.

4.3 Environment Development

The environment is developed using the C++ programming language and the Qt Framework. According to recent studies, C++ is one of the fastest programming languages and offers fine-grained control over system resources [13]. The Qt Framework has been chosen to implement the graphical user interface. Thanks to its built-in signal and slot mechanism [14], the framework enables rapid development of user interactions with the application. It also offers a rich set of graphical tools, which is a significant advantage for visualizing the dependency graph.

The first step in the development is to identify the dataset collected during program execution, as well as to determine a structure for storing this data. The JSON format is selected for data storage due to its human-readability and broad tool support. The collected data is categorized into two main types: general execution data and data required to reconstruct the dependency graph.

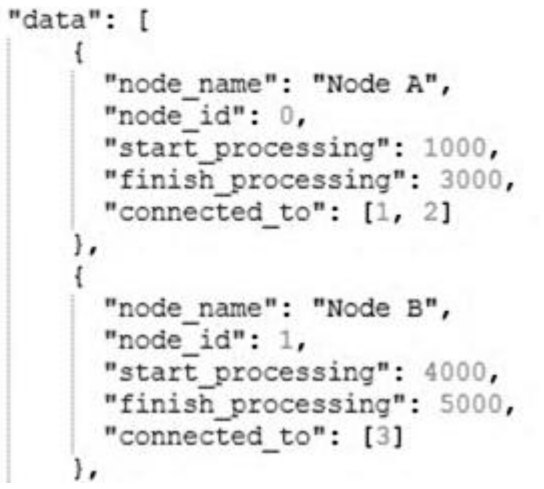
General execution data includes the total execution time of the program and the active time of each allocated thread. Only the durations in which threads were processing subtasks (i.e., performing useful work) are counted (Fig. 1).



```
"stats": {
    "total_time": 7000,
    "workers": [
        7000,
        2000
    ]
},
```

Fig. 1. Example of collected general execution data.

To reconstruct the dependency graph, data for each graph node is recorded in the following format: a unique node identifier, the name of the node (or subtask, for easier interpretation of the graph’s structure), the list of identifiers corresponding to the nodes that this node connects to via outgoing edges, the start time of the node’s processing, and the completion time of that processing (Fig. 2). All this information is collected automatically during the execution of a multithreaded program implemented using the task execution control method based on a predefined dependency graph.



```
"data": [
    {
        "node_name": "Node A",
        "node_id": 0,
        "start_processing": 1000,
        "finish_processing": 3000,
        "connected_to": [1, 2]
    },
    {
        "node_name": "Node B",
        "node_id": 1,
        "start_processing": 4000,
        "finish_processing": 5000,
        "connected_to": [3]
    },
```

Fig. 2. Example of collected graph data.

The next challenge addressed during the development of the environment was the visualization of the dependency graph itself. A wide range of algorithms for graph visualization is available to solve this type of task [15]. Given that, according to the method’s definition, the dependency graph is directed and acyclic, the most suitable approach for its visualization is the hierarchical graph layout method, also known as the Sugiyama framework [16] (Fig. 3).

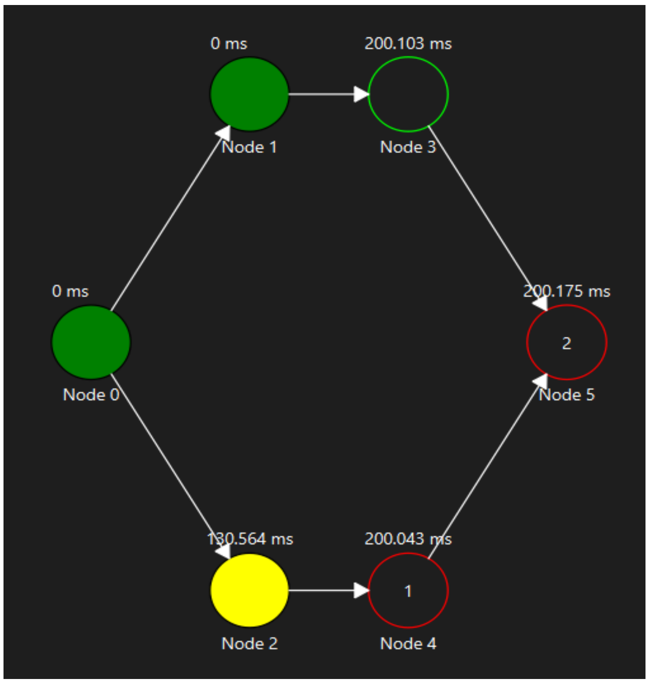


Fig. 3. Visualizing the dependency graph within the environment using the Sugiyama layout method.

To visualize the program execution process based on the dependency graph and to enable its simulation, a dedicated class named ‘`GraphController`’ is developed. This class contains a complete representation of the graph in the form of an array of ‘`GraphNode`’ objects, as well as the logic for updating the graph state according to simulation time (see Figure 3).

During the program execution, each ‘`GraphNode`’ object displays the number of unresolved dependencies that prevent the corresponding subtask from being executed. A ‘`GraphNode`’ can be in the following states:

– blocked (marked with a red outline) if none of the dependencies of this node have been fulfilled,

– partially unblocked (marked with a yellow outline) if some, but not all, dependencies of the node have been fulfilled,

– ready (marked with a green outline) if all dependencies have been resolved, and the subtask associated with this node is queued for execution,

– executing (marked with a solid yellow fill) if the subtask associated with this node is currently being executed,

– processed (marked with a solid green fill) if the subtask associated with this node has completed execution.

The ‘`GraphController`’ operates in two distinct modes: step-by-step playback and simulation mode. Depending on the selected mode, ‘`GraphController`’ utilizes either ‘`ReplayGraphNode`’ or ‘`SimulationGraphNode`’ objects, since these types implement different logic for updating the state of each graph node based on time progression.

In playback mode, the environment reproduces the program execution according to data previously recorded in a file. This allows the developer to visually analyze the graph state and identify performance bottlenecks that negatively impact execution speed.

In simulation mode, the execution of the graph is emulated using the computation mechanism described in the method for managing multithreaded task execution via a predefined dependency graph. In this mode, the user can specify any number of threads for the simulation, regardless of the actual configuration of the system. Before starting the simulation, the user may choose whether to visualize the simulation process or to focus solely on the resulting statistical data.

The execution statistics include the following metrics:

– total execution time in milliseconds,

– efficiency coefficient, calculated as the average utilization of all threads,

– per-thread statistics, including work time in milliseconds, idle time in milliseconds, ratio of work time to total program execution time.

5. Results of the investigation of the environment for tuning parameters of the multithreaded program

As a result of the investigation, an environment for tuning parameters of the multithreaded program is proposed and developed. To verify that the environment produces accurate and valid results, an experiment is required.

For the experimental investigation of the accuracy of multithreaded program simulation, a method for controlling the execution of tasks within a multithreaded application according to the given dependency graph was implemented in C++, as described in [1].

To test the simulation on graphs of varying sizes, an algorithm for generating random directed acyclic graphs was also added. The generated graph adheres to the following requirements:

– the number of graph vertices equals a predefined quantity N,

– the number of edges is random and ranges between N and $5N$ (this constraint prevents an excessive number of dependencies that would otherwise make parallel execution practically impossible),

– the generated graph is connected.

Each vertex of the graph corresponds to a subtask that simulates a time delay of approximately 20±1 milliseconds. In this case, the random distribution reproduces the variability in the execution time of the same code segment across different runs.

A busy waiting approach is used to simulate the time delay. The rationale behind using busy waiting lies in the fact that the thread remains actively engaged in computation, preventing operating system optimization mechanisms from reallocating resources away from inactive threads. Otherwise, the experimental results could be distorted [17].

Graphs with the following parameters were generated for testing:

– 100 vertices, 406 edges,

– 250 vertices, 1045 edges,

– 500 vertices, 1122 edges,

– 1000 vertices, 3551 edges.

For each generated graph, the program was executed using 2, 4, 6, 8, and 10 threads.

To evaluate the accuracy of the simulation, both the absolute execution time and the efficiency of computational resource utilization were compared. The efficiency indicators of resource utilization are calculated using the following formula:

$$k = \frac{\sum_{i=0}^{N-1} t_i}{N \cdot T}, \tag{1}$$

where k is the efficiency coefficient of computational resource utilization, N is the number of program threads, t_i is the execution time of the i-th thread (only the time spent on subtask processing is considered), T is the total runtime of the program.

As a result of the research, simulation accuracy of 93.97% was achieved for the program execution time, and 95.55% for the calculated efficiency of system resource utilization. These results support the conclusion that the simulation algorithm operates correctly.

6. Discussion of the results of developing the environment for tuning parameters of the multithreaded program

6.1 Description of the software demonstration

To demonstrate the practical application of the developed simulation environment, the program was created that implements the Canny edge detection algorithm [18] in C++, based on the method for managing the execution of tasks in a multithreaded program via a predefined dependency graph [1].

For parallel image processing, the image is divided into individual horizontal strips, each of which is processed independently. However, during most stages of the Canny algorithm, a common issue arises concerning the handling of border pixels at the edges of each strip. This problem is effectively addressed by the task execution management method using a dependency graph. It enables the treatment of border pixel processing as a separate subtask, with its dependencies configured in such a way that the subtask is only executed after all adjacent strips have completed their processing.

6.2 Application of the environment for tuning parameters of a multithreaded program

It is assumed that the developed implementation of the Canny algorithm will be deployed on a cloud computing platform as a component of a larger software system. In this scenario, the developer must determine the appropriate number of computational resources to allocate for executing this program. This decision is critical: insufficient resources can degrade the overall performance of the product, while allocating excessive resources results in unnecessary financial costs, especially if the program cannot utilize those resources efficiently.

To address this issue, the proposed environment for tuning the parameters of a multithreaded program based on a dependency graph can be employed.

Suppose the input images for Canny processing are of size 2048×2048 pixels. In this case, dividing the image into 16 horizontal strips is considered optimal.

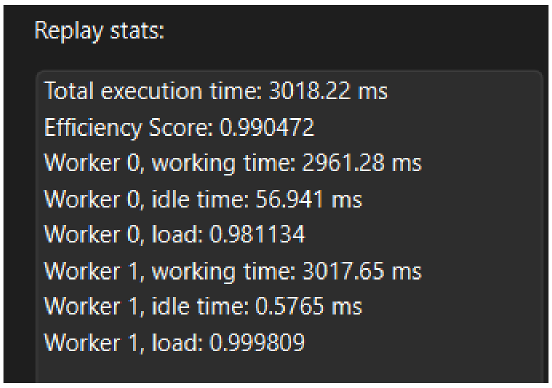
Additionally, we assume the following non-functional requirements for the deployed application:

– the processing time for a single image must not exceed 2 seconds,

– the number of logical CPU cores allocated should not exceed 6.

It is further assumed that, for the initial experiment, the program was executed in a 2-thread configuration, and its execution statistics were recorded (see Figure 4). It was observed that, although the efficiency of computational resource utilization was nearly 100%, the program's execution time exceeded 2 seconds. Therefore, the configuration using 2 logical cores does not satisfy the program's non-functional requirements.

To determine the optimal resource configuration, the developer would typically need to repeat the experiment with different numbers of logical cores, compare the resulting metrics, and draw conclusions about which configuration best meets the non-functional constraints. However, conducting such experiments is resource-intensive. In addition to the computational overhead, performing repeated tests incurs additional time and financial costs, especially when using cloud infrastructure.



```
Replay stats:

Total execution time: 3018.22 ms
Efficiency Score: 0.990472
Worker 0, working time: 2961.28 ms
Worker 0, idle time: 56.941 ms
Worker 0, load: 0.981134
Worker 1, working time: 3017.65 ms
Worker 1, idle time: 0.5765 ms
Worker 1, load: 0.999809
```

Fig. 4. Statistics 2-thread execution of the program.

The proposed tuning environment for multithreaded programs addresses this challenge by enabling the simulation of program execution under alternative thread configurations based on the results of a single empirical run. This significantly reduces the resource demands of the optimization process.

In this case, simulation results were generated using the empirical data obtained from the experiment with two threads, and the outcomes are summarized in Table 1.

Table 1. Simulation results

Number of threads	Execution time, ms	Efficiency indicator of resource utilization	Satisfies non-functional requirements
3	2030	0.9818	No
4	1529.57	0.9772	Yes
5	1240.93	0.9636	Yes
6	1044.95	0.9536	Yes

The non-functional requirements are satisfied by the results obtained for four, five, and six threads. The fastest result was achieved with six threads, while the highest efficiency of computational resource utilization was demonstrated by the four-thread configuration. Thus, we can conclude that according to the simulation, the fastest configuration would be the use of six threads, and the most efficient and most suitable for meeting the non-functional requirements – four threads.

For the obtained results, validation will be performed by executing the actual program using four and six threads. The validation results are presented in Table 2.

Table 2. Simulation results validation

Number of threads	Execution time, ms			Efficiency indicator of resource utilization		
	Program	Simulation	Accuracy	Program	Simulation	Accuracy
4	1493.87	1529.57	97.6%	0.9749	0.9772	99.8%
6	996.218	1044.95	95.1%	0.9572	0.9536	99.2%

Thus, it is concluded that the environment for tuning the parameters of a multithreaded program provided entirely accurate results based on the simulation, while significantly reducing the resource intensity of the process of investigating these parameters.

Conclusion

To reduce the resource cost of tuning the parameters of a multithreaded program, an environment for tuning parameters of a multithreaded program developed using a dependency graph is created. The environment allows to significantly reduce the resource cost of the process of tuning the parameters of a multithreaded program. This was proved by demonstrating the application of the developed environment to the practical task. The validity of the results obtained through simulation is confirmed by comparing them with those obtained from real program executions.

An analysis of existing solutions for tuning parameters of a multithreaded program was conducted. The issue of intense resource consumption of existing methods was identified. As a solution, an environment for tuning parameters of a multithreaded program developed using a dependency graph is proposed and developed. A general overview of the environment is provided, along with a detailed description of its development process and implementation specifics.

The accuracy of the developed simulation algorithm, used by the environment, is evaluated, confirming its correctness and efficiency. For the test data, simulation accuracy of 93.97% was achieved for the program execution time, and 95.55% for the calculated efficiency of system resource utilization. These results support the conclusion that the simulation algorithm operates correctly.

References

[1] K. P. Nesterenko and I. V. Stetsenko, “Method of managing the execution of tasks of a multithreaded program according to a given dependency graph,” in *Problems in Programming*, no. 2–3, pp. 239–246, 2024. https://doi.org/10.15407/pp2024.02-03.239.

[2] S. Borkar and A. Chien, “The Future of Microprocessors,” in *Communications of the ACM*, vol. 54 (5), pp. 67–77, 2011. https://doi.org/10.1145/1941487.1941507.

[3] R. Rakvic, Q. Cai, J. González, G. Magklis, P. Chaparro, and A. González, “Thread-management techniques to maximize efficiency in multicore and simultaneous multithreaded microprocessors,” in *ACM Transactions on Architecture and Code Optimization*, vol. 7 (2), article no. 9, 2010. https://doi.org/10.1145/1839667.1839671.

[4] Tharwani, Jay & Purkayastha, Arnab. (2024). Cost-Performance Evaluation of General Compute Instances: AWS, Azure, GCP, and OCI. http://doi.org/10.48550/arXiv.2412.03037.

[5] J. Cui, J. L. Bordim, K. Nakano, T. Hayashi, and N. Ishii, “Multithreaded Parallel Computer Model with Performance Evaluation,” in *Lecture Notes in Computer Science*, vol. 1800, pp. 155–160, 2000. https://doi.org/10.1007/3-540-45591-4_20.

[6] I. V. Stetsenko, O. Dyfuchyna, “Thread Pool Parameters Tuning Using Simulation,” *Advances in Intelligent Systems and Computing*, vol. 938, pp. 78-89, 2020. https://doi.org/10.1007/978-3-030-16621-2_8.

[7] CPN IDE. “CPN IDE Documentation”. Accessed: June 21, 2025. [Online]. Available: https://cpnide.org/category/documentation/.

[8] “Visual Studio Profiling Documentation,” Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/visualstudio/profiling. Accessed: Jun. 21, 2025.

[9] “Unreal Engine Insights Documentation,” Epic Games. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-insights-in-unreal-engine. Accessed: June 21, 2025.

[10] “RAD Game Tools. Telemetry Performance Visualization System,” [Online]. Available: https://www.radgametools.com/telemetry.htm. Accessed: June 21, 2025.

[11] M. A. Khan, “Improving performance through deep value profiling and specialization with code transformation,” in *Computer Languages, Systems and Structures*, vol. 37, pp. 193–203, 2011, https://doi.org/10.1016/j.cl.2011.08.001.

[12] Yang, Chen & Zhang, Jing & Xie, Xiguo & Sun, Jianpeng. (2024). Research on Application Performance Analysis Methods for Scientific Computing. 429–436. http://doi.org/10.1145/3690931.3691003.

[13] “Debian. The Computer Language Benchmarks Game (CLBG),” [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/. Accessed: June 21, 2025.

[14] Q.-X. Wu and J. Ou, “QT Programming Technology and Application with Linux,” in *Advances in Intelligent and Soft Computing*, vol. 114, pp. 573–578, 2012, https://doi.org/10.1007/978-3-642-03718-4_71.

[15] S. Di Bartolomeo, T. Crnovrsanin, D. Saffo, E. Puerta, C. Wilson, and C. Dunne, “Evaluating Graph Layout Algorithms: A Systematic Review of Methods and Best Practices,” in *Computer Graphics Forum*, 43 (6), 15073, 2024, https://doi.org/10.1111/cgf.15073.

[16] C. Bachmaier, “A radial adaptation of the Sugiyama framework for Visualizing Hierarchical Information,” in *IEEE Transactions on Visualization and Computer Graphics*, 13 (3), pp. 583–594, 2007, https://doi.org/10.1109/TVCG.2007.1000.

[17] J. Blieberger, B. Burgstaller, and B. Scholz, “Busy Wait Analysis,” in *Lecture Notes in Computer Science*, vol. 2655, pp. 142–152, 2003, https://doi.org/10.1007/3-540-44947-7_10.

[18] R. Liu and J. Mao, “Research on Improved Canny Edge Detection Algorithm,” in *MATEC Web of Conferences*, vol. 232, 03053, 2018, https://doi.org/10.1051/matecconf/201823203053.

УДК 004.42

СЕРЕДОВИЩЕ ДЛЯ НАЛАШТУВАННЯ ПАРАМЕТРІВ БАГАТОПОТОКОВОЇ ПРОГРАМИ, РОЗРОБЛЕНОЇ НА ОСНОВІ ГРАФУ ЗАЛЕЖНОСТІ

Костянтин Нестеренко
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського», Київ, Україна
https://orcid.org/0000-0003-3921-4324

Інна Стеценко
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського», Київ, Україна
https://orcid.org/0000-0002-4601-0058

З появою багатоядерних центральних процесорів застосування багатопотоковості стало найбільш поширеною практикою для збільшення швидкодії виконання програм. Проте розробка багатопотокової програми є досить складним процес. З метою спрощення цього процесу та покращення швидкодії результуючої програми, часто використовуються різноманітні методи управління виконанням програми у потоках.

Одним з таких методів є метод управління виконанням задач багатопотокової програми за заданим графом залежностей. Метод дозволяє суттєво зменшити ресурсоємність процесу розробки програми, а також підвищити швидкодію розробленої програми за рахунок використання неблокуючого підходу до багатопотокового програмування.

Проте актуальною залишається проблема ефективності використання обчислювального ресурсу, вирішити яку вдається тільки ретельною розробкою паралельних обчислень. Зокрема, визначення параметрів багатопотокової програми, що забезпечують найбільш ефективне використання обчислювального ресурсу, є ресурсовитратним та складним завданням навіть для висококваліфікованого фахівця.

У даному дослідженні розглянуті існуючі підходи до налаштування параметрів багатопотокової програми, що забезпечують найбільш ефективне її виконання. Запропоновано використовувати середовище для налаштування параметрів багатопотокової програми, на основі методу управління виконанням задач багатопотокової програми за заданим графом залежностей. Експериментально було доведено точність результатів обчислення ефективності використання обчислювального ресурсу, отриманих за допомогою середовища. На прикладі описано процес застосування середовища для розробки багатопотокової програми. Використання середовища сприяє також спрощенню процесу налаштування параметрів багатопотокової програми.

Ключові слова: програмне забезпечення, багатопотоковість, обчислювальні ресурси, граф залежностей, паралельне програмування мовою C++.