https://doi.org/10.31891/csit-2025-2-12
UDC 004.41+004.43:519.6

ZHULKOVSKYI Oleg
Dniprovsky State Technical University
ZHULKOVSKA Inna
University of Customs and Finance
VOKHMIANIN Hlib
Dniprovsky State Technical University
TKACH Anastasiia
Dniprovsky State Technical University

COMPARATIVE ANALYSIS OF COMPUTATIONAL PERFORMANCE OF MODERN PROGRAMMING LANGUAGES

The study is dedicated to the comparative analysis of the computational performance of modern programming languages in the implementation of numerical methods for solving boundary value problems in mathematical physics. The central focus of the research is the Thomas algorithm – an efficient numerical method for solving systems of linear algebraic equations with a tridiagonal matrix. The research methodology is based on a unified implementation of the Thomas algorithm for each examined programming language, ensuring identical algorithmic logic. Experimental testing was conducted on systems with sizes ranging from 10^5 to 1.5×10^7 elements for programming languages including C, C++, C#, Java, JavaScript, Go, and Python, which represent different paradigms and approaches to computation. The obtained results demonstrate significant differences in the performance of various programming languages. It was established that low-level compiled languages exhibit the highest execution speed, especially for large problem sizes. In contrast, interpreted languages show significantly lower performance, which becomes more pronounced as the computational workload increases.

The study experimentally confirmed the impact of compiler optimization modes on performance, revealing differences of up to 70% depending on the language and optimization level. The scientific novelty of this work lies in the comprehensive investigation of programming language performance in the context of numerical modeling by comparing their characteristics when solving mathematical problems. Future research will include an in-depth study of the impact of processor architecture, compiler optimization mechanisms, and runtime environment implementation on the performance of computational algorithms, as well as an expansion of the range of numerical methods and programming languages analyzed.

Keywords: programming language performance, SLAE, Thomas algorithm, optimization levels.

ЖУЛЬКОВСЬКИЙ Олег
Дніпровський державний технічний університет
ЖУЛЬКОВСЬКА Інна
Університет митної справи та фінансів
ВОХМЯНІН Гліб, ТКАЧ Анастасія
Дніпровський державний технічний університет

ПОРІВНЯЛЬНИЙ АНАЛІЗ ОБЧИСЛЮВАЛЬНОЇ ШВИДКОДІЇ СУЧАСНИХ МОВ ПРОГРАМУВАННЯ

Дослідження присвячене порівняльному аналізу обчислювальної швидкодії сучасних мов програмування при реалізації чисельних методів розв'язання крайових задач математичної фізики. Центральним об'єктом дослідження виступає метод прогонки – ефективний чисельний алгоритм розв'язання систем лінійних алгебраїчних рівнянь з тридіагональною матрицею. Методологія дослідження базується на уніфікованій реалізації методу прогонки для кожної досліджуваної мови з ідентичною алгоритмічною логікою. Експериментальні випробування проведено на системах розмірністю від 10^5 до $1,5 \times 10^7$ елементів для мов програмування C, C++, C#, Java, JavaScript, Go, Python, які репрезентують різні парадигми та підходи до виконання обчислень. Отримані результати демонструють суттєві відмінності у продуктивності різних мов програмування. Встановлено, що компільовані мови низького рівня демонструють найвищу швидкодію, особливо при великих розмірностях задач. Натомість інтерпретовані мови мають значно нижчу продуктивність, що відстежується при збільшенні обсягу обчислень. Експериментально підтверджено вплив оптимізаційних режимів компіляції на продуктивність, демонструючи різницю до 70% залежно від мови та рівня оптимізації. Наукова новизна роботи полягає в комплексному дослідженні продуктивності різних мов програмування в контексті чисельного моделювання шляхом порівняння їхніх характеристик при розв'язанні математичних задач. Подальші дослідження включають поглиблене вивчення впливу архітектури процесора, механізмів оптимізації компіляторів та особливостей реалізації середовищ виконання на продуктивність обчислювальних алгоритмів, а також розширення спектру досліджуваних чисельних методів та мов програмування.

Ключові слова: швидкодія мови програмування, СЛАР, метод прогонки, рівні оптимізації.

Introduction

In the context of rapid development of software and computing technology, the issue of intensifying computational processes is becoming increasingly relevant, as the speed of algorithm execution directly affects data processing efficiency and system performance [1]. The diversity of modern systems and programming languages, their interaction with hardware resources, memory management, compilation processes, and other factors highlight the importance of selecting an appropriate language for solving computational problems. A particularly critical criterion in choosing a programming language for computational modeling tasks is the execution speed of programs.

Thus, researchers face the question of which programming language is the most effective for implementing numerical algorithms used in computer models.

In computational modeling tasks that ultimately reduce to the numerical solution of systems of linear algebraic equations (SLAE) with a tridiagonal matrix, the Tri-Diagonal Matrix Algorithm (TDMA) is frequently used. TDMA is specifically applied to solving SLAEs arising from the discretization of differential equations using the finite difference or finite element methods for boundary value problems. It is one of the most efficient algorithms, ensuring SLAE solutions in linear time [2].

Despite its algorithmic efficiency, the practical performance of TDMA can vary significantly depending on the programming language used for its implementation.

The aim of this study is to identify the most efficient programming languages in terms of execution speed for the TDMA implementation. For comparative analysis, the study will examine modern high-ranking [3] programming languages: C, C++, C#, Python, Java, JavaScript, and Go.

Related works

The performance of computational processes is determined not only by the specifics of their algorithmic implementation but also by other factors, including the choice of programming language. Programming languages differ in their paradigms, memory management mechanisms, levels of abstraction, and execution models, all of which directly impact the speed and resource efficiency of implemented algorithms [1, 4]. The relationship between programming languages and execution efficiency becomes increasingly significant given the continuous growth of data volumes that require complex processing. The evolution of computing architectures, including multi-core processors, graphics accelerators, and specialized computational devices [5], adds an additional layer of complexity, as different languages exhibit varying degrees of efficiency in hardware optimization. Consequently, the same algorithm, when implemented in different programming languages, may exhibit different execution times.

For example, the discrete wavelet transform (DWT) method for audio signal analysis demonstrates varying execution times when implemented in different programming languages [1]. The results highlight the advantages and superior performance of C and C++ in digital signal processing tasks.

In another context, a comparison of PHP, Python, Node.js, and Golang for API development in cloud environments has shown [4] that Golang provides the best scalability and performance under high loads, whereas Node.js performs well in systems with medium workloads.

Reference [6] presents a performance analysis of JSON parsers in Java, Python, C#, JavaScript, and PHP, examining their efficiency in terms of parsing speed and resource consumption within their respective native environments. The evaluation was conducted using JSON test files with varying levels of nesting and data volumes. The study utilized monitoring tools on the Windows 10 operating system to analyze performance, enabling the identification of language-specific parser implementations in the context of semi-structured data processing.

Interpreted languages such as Python offer greater flexibility and faster development cycles, thereby improving developer productivity. Although interpreted languages are generally slower than compiled ones [7], advancements such as just-in-time (JIT) compilers and transpilers have significantly narrowed this performance gap [8]. A study comparing three compiled languages – Fortran, C++, and Java – and three interpreted languages – Python, MATLAB, and Julia – was conducted based on their popularity and technical advantages. These six languages were evaluated and compared in terms of capabilities, performance, and ease of use through the implementation of idiomatic solutions to classical astrodynamics problems. The results confirmed that compiled languages still provide the highest execution performance, but JIT-compiled dynamic languages have achieved competitive speed levels and offer an attractive trade-off between numerical efficiency and developer productivity.

A comparative analysis of Python and Scala for big data processing was conducted using Apache Spark – an open-source in-memory cluster computing system designed for high-speed processing [9]. The study concluded that both languages are suitable for Apache Spark, but the choice depends on project-specific requirements, balancing development convenience, performance, and data integration efficiency.

Reference [10] also compares the performance of Scala and Java in the Apache Spark MLlib environment through a series of tests involving various machine learning methods. The experiments demonstrated that Scala outperforms Java by 10–20% in performance, depending on the algorithm's characteristics.

Reference [11] analyzes the performance of C, Python, MATLAB, and LabVIEW in instrumentation automation tasks. The results indicate that C exhibits minimal resource consumption and optimal performance for small data volumes, whereas Python is advantageous for interface setup and efficient memory usage. MATLAB delivers the fastest processing for large datasets, while LabVIEW surpasses other tools in real-time control tasks and maintains stable performance in graphical rendering. Similar to the findings in [9], language selection depends on task-specific requirements – C is optimal for resource-constrained systems, MATLAB is suited for complex computations, LabVIEW is ideal for real-time control, and Python excels in multi-device integration.

A systematic review of WebAssembly (WASM) and JavaScript performance in various aspects (execution time, memory usage, and energy consumption) demonstrated [12] that WASM is more efficient in lightweight applications due to its faster execution and lower energy consumption. However, in more complex applications, JavaScript exhibits lower resource consumption and higher execution speed.

A comparative performance analysis of MicroPython (a Python implementation in C for microcontrollers) and C was conducted on STM32 and ESP32 microcontrollers [13]. The study evaluated memory allocation speed and the execution efficiency of cryptographic algorithms such as SHA-256 and CRC-32. The results indicated that despite certain limitations, MicroPython can be an effective tool for low-cost microcontrollers when appropriate optimizations are applied.

Reference [14] further examines the performance of C/C++, MicroPython, Rust, and TinyGo on the ESP32 microcontroller in IoT applications. The study focused on data and signal processing algorithm execution speed, as well as development convenience. The results revealed the advantages and limitations of each language depending on specific requirements and application scenarios. Implementations in C/C++ were the fastest in most cases, followed by TinyGo and Rust, while MicroPython applications were significantly slower. Thus, C/C++, TinyGo, and Rust are better suited for scenarios where execution time and response time are critical, whereas Python offers a faster and less complex development process for less stringent system requirements.

A comparative analysis of Go, Java, and Python in decision-support processes for Industry 4.0 was conducted in [15]. The study examined decision tree algorithms based on entropy heuristics, evaluating parameters such as memory usage, CPU utilization, and computation time to determine the suitability of these languages for industrial solutions in the Industry 4.0 framework.

Programming languages may exhibit different execution times for algorithms implemented synchronously, asynchronously, or in parallel. In parallel metaheuristics commonly used for solving NP-hard optimization problems, a comparison of Chapel, Julia, and Python demonstrated [16] that none of these languages outperform C combined with OpenMP in performance. However, they offer a trade-off between execution speed and development convenience.

Reference [17] compares structured approaches to parallelism in Java and Kotlin, analyzing their performance, real-world adaptability, and optimization capabilities for multi-threaded applications. Both languages operate on the Java Virtual Machine (JVM), inherently supporting traditional thread-based concurrency. However, Kotlin includes lightweight coroutines for parallelism, whereas Java’s virtual threads, introduced in Project Loom, remain experimental. A review of recent scientific studies highlights the multifactorial dependence of computational performance on programming language choice, driven by differences in paradigms, memory management mechanisms, abstraction levels, and execution models. Studies demonstrate that lower-level languages, such as C and C++, provide the highest efficiency in digital signal processing and resource-constrained systems. For web development and API performance, efficiency varies – Golang offers optimal scalability under high loads, while Node.js is a suitable choice for medium workloads. In big data processing, the choice between Python and Scala for Apache Spark depends on the balance between development convenience and integration efficiency. Comparative studies of WebAssembly and JavaScript reveal context-dependent efficiency: WASM performs better in simple applications, while JavaScript excels in more complex ones. In microcontroller-based systems, C/C++, TinyGo, and Rust confirm their advantages in execution time and response speed, whereas MicroPython provides a trade-off between development speed and efficiency.

A review of recent scientific studies highlights the multifactorial dependence of computational performance on programming language choice, driven by differences in paradigms, memory management mechanisms, abstraction levels, and execution models. Studies demonstrate that lower-level languages, such as C and C++, provide the highest efficiency in digital signal processing and resource-constrained systems. For web development and API performance, efficiency varies – Golang offers optimal scalability under high loads, while Node.js is a suitable choice for medium workloads. In big data processing, the choice between Python and Scala for Apache Spark depends on the balance between development convenience and integration efficiency. Comparative studies of WebAssembly and JavaScript reveal context-dependent efficiency: WASM performs better in simple applications, while JavaScript excels in more complex ones. In microcontroller-based systems, C/C++, TinyGo, and Rust confirm their advantages in execution time and response speed, whereas MicroPython provides a trade-off between development speed and efficiency. Parallel computing support is also a significant factor affecting performance. Traditional C with OpenMP [18] retains its dominant position over higher-level but more developer-friendly languages such as Chapel, Julia, and Python.

In the context of Industry 4.0 and decision-support systems, key selection criteria for programming languages include memory usage, CPU utilization, and computational time. Differences in parallelism models also contribute to performance variability. Thus, selecting a programming language for a specific task should be based on a comprehensive analysis of execution efficiency, development convenience, and application-specific requirements, as supported by systematic empirical research.

To achieve the research objective, this study aims to perform a comparative analysis of the computational efficiency of TDMA implementations using the prominent programming languages C, C++, C#, Python, Java, JavaScript, and Go.

Materials and Methods

For conducting a comparative analysis of the computational efficiency of different programming languages, TDMA was chosen, as described in [19].

The mathematical model of the problem for numerically solving a system of *n* linear algebraic equations is as follows:

$$\begin{aligned}
&a_i x_{i-1} - c_i x_i + b_i x_{i+1} = -f_i,\\
&a_i \neq 0,\, b_i \neq 0,\, i = 1...n;\\
&a_1 = 0,\, b_n = 0;\\
&|c_1| \geq |b_1|;\\
&|c_n| \geq |a_n|;\\
&|c_i| > |a_i| + |b_i|,\, i = 2...n-1,
\end{aligned}$$

where the last three inequalities represent the diagonal dominance conditions, ensuring the numerical stability of the method.

The pseudocode for the TDMA solution of a tridiagonal system of linear algebraic equations using the Double Sweep Method can be presented as follows:

TDMA: pseudo code (Double Sweep Method)

```
1.  p = n / 2
2.  // forward pass
3.  For i = 1 to p do // right sweep
4.     den = c[i] - a[i] * alfa[i]
5.     alfa[i+1] = b[i] / den
6.     beta[i+1] = (a[i] * beta[i] + f[i]) / den
7.  End for
8.
9.  For i = n downto p do // left sweep
10.    den = c[i] - b[i] * ksi[i+1]
11.    ksi[i] = a[i] / den
12.    eta[i] = (b[i] * eta[i+1] + f[i]) / den
13. End for
14.
15. // conjugation of solutions
16. x[p] = (alfa[p+1] * eta[p+1] + beta[p+1]) / (1 – alfa[p+1] * ksi[p+1])
17.
18. // backward pass
19. For i = p-1 downto 1 do // right sweep
20.    x[i-1] = alfa[i+1] * x[i+1] + beta[i+1]
21. End for
22.
23. For i = p to n-1 do // left sweep
24.    x[i+1] = ksi[i+1] * x[i] + eta[i+1]
25. End for
```

At the beginning of the algorithm (line 1), the splitting point *p* is set, dividing the system of dimension *n* in half. In lines 3–7, the forward sweep of the right pass is performed to compute the sweep coefficients α, β for the first half of the system. In lines 9–13, the forward sweep of the left pass is executed to compute the sweep coefficients ξ, η for the second half of the system, respectively. The reconciliation of both solution parts at point *p* is implemented in line 16. After coupling, the backward sweeps are performed: in lines 19–21 for the right pass and in lines 23–25 for the left pass, respectively.

Experiments

All experiments were conducted on a single computer with the following specifications:

– CPU: AMD Ryzen 5 350OU, 2100 MHz, 4 cores, 8 threads;

– RAM: Goodram DDR4 (4 GB, 2666 MHz) × 2 = 8 GB;

– OS: Microsoft Windows 11 Pro.

The experiments were conducted for the system size n=10^5–1.5×10^7.

Seven programming languages were chosen for the study, representing different programming paradigms and approaches to compilation and interpretation (Table 1).

Table 1

Characteristics of the studied programming languages

Language	Classification	Typing	Brief description
C	Compiled	Static	A mid-level language with a minimal level of abstraction. Implemented using

			the standard GCC compiler [20].
C++	Compiled	Static	A language with support for object-oriented and generic programming. Implemented using the standard G++ compiler [20].
C#	Compiled	Static	A language for the .NET platform, using intermediate code (IL) and a virtual machine (CLR) [21].
Java	Compiled-Interpreted	Static	An object-oriented programming language that uses bytecode, executed by the Java Virtual Machine (JVM) [22].
JavaScript (JS)	Interpreted	Dynamic	A multi-paradigm language, implemented using various engines, including Node.js for server-side execution [23].
Python	Interpreted	Dynamic	A multi-paradigm language, implemented with the standard CPython interpreter [24].
Golang (Go)	Compiled	Static	A language with built-in support for concurrent programming and fast compilation. Developed to improve development productivity [25].

The methodology for measuring execution time is adapted to the specifics of each programming language using the following approaches:

– The clock() function from the time.h library for the C language [20];
– The use of std::chrono::steady_clock for high-precision measurement in C++ [20];
– The application of the Stopwatch class from the System.Diagnostics namespace in C# [21];
– Measurement using System.nanoTime() in Java [22];
– The performance.now() function from the high-precision time measurement API in JavaScript [23];
– The time.perf_counter() function, providing the highest available resolution in Python [24];
– The time.Now() function and duration measurement methods in Go [25].

During the experiments, only the execution time of the algorithmic part of the program was measured, excluding the time for initializing the execution environment, loading the interpreter, or allocating memory for the initial data structures.

Compilation and execution of the programs were performed using the built-in optimization features of each programming language.

To investigate the impact of built-in optimization efficiency, additional experiments were conducted for C-like languages in Debug x64 mode (no optimization, -Od) and Release x64 mode (optimization with a focus on execution speed, -O2).

Results

The obtained results are summarized in Table 2 and Table 3 and presented as graphs showing the relationship between execution time and the system size for different programming languages (Fig. 1–3).

Table 2

Results of computational experiments for Python, Java, JavaScript, and Go with built-in optimization

SLAE order	Computation time, s			
	Python	Java	JavaScript	Go
1×10^5	0.1030	0.0193	0.0207	0.0020
2×10^5	0.2064	0.0258	0.0239	0.0050
3×10^5	0.3278	0.0357	0.0330	0.0070
4×10^5	0.4283	0.0359	0.0427	0.0098
5×10^5	0.4607	0.0386	0.0430	0.0118
6×10^5	0.5906	0.0396	0.0520	0.0142
7×10^5	0.7272	0.0416	0.0488	0.0180
8×10^5	0.8575	0.0490	0.0547	0.0208
9×10^5	0.9914	0.0479	0.0609	0.0244
1×10^6	1.2253	0.0490	0.0653	0.0280
2.5×10^6	2.8130	0.0850	0.1563	0.0589
5×10^6	5.6513	0.1397	0.3152	0.1202
1×10^7	10.2515	0.2786	0.6183	0.2498
1.5×10^7	14.9313	0.4496	0.9861	0.4055

The comparative analysis of interpreted languages (Python, JavaScript) and compiled languages (Java, Go, C, C++, C#) revealed a nonlinear dependence of execution time on system size, which follows an exponential pattern (Fig. 2). For Python and JavaScript, there is significantly higher sensitivity to the increase in system size. Specifically, when transitioning from a system size of 10^5 to 10^7 the execution time for Python increases approximately 145 times (from 0.103 s to 14.9313 s), and for JavaScript, it increases more than 30 times (from 0.0207 s to 0.9861 s).

On the other hand, compiled languages show a fundamentally different pattern. Go demonstrates almost a linear relationship between execution time and system size. C and C++ exhibit similar results with minimal execution time overhead. Notably, for the maximum system size of 1.5×10^7 Python requires nearly 15 seconds of computation, whereas Go takes only 0.4055 seconds, Java 0.4496 seconds, and C++ about 0.3637 seconds (Fig. 3).

Table 3

Results of computational experiments for C-like languages in different optimization modes

SLAE order	Computation time, s					
	C (-Od)	C (-O2)	C++ (-Od)	C++ (-O2)	C# (-Od)	C# (-O2)
1×10^5	0.0010	0.0010	0.0023	0.0030	0.0050	0.0040
2×10^5	0.0050	0.0030	0.0062	0.0045	0.0116	0.0067
3×10^5	0.0080	0.0060	0.0073	0.0056	0.0143	0.0088
4×10^5	0.0110	0.0060	0.0106	0.0115	0.0191	0.0115
5×10^5	0.0150	0.0090	0.0162	0.0140	0.0223	0.0165
6×10^5	0.0320	0.0090	0.0179	0.0189	0.0339	0.0183
7×10^5	0.0190	0.0140	0.0212	0.0211	0.0399	0.0203
8×10^5	0.0250	0.0140	0.0245	0.0246	0.0406	0.0220
9×10^5	0.0260	0.0180	0.0279	0.0277	0.0443	0.0266
1×10^6	0.0290	0.0150	0.0373	0.0293	0.0533	0.0314
2.5×10^6	0.0670	0.0420	0.0631	0.0632	0.1432	0.0770
5×10^6	0.1170	0.0780	0.1213	0.1113	0.2553	0.1684
1×10^7	0.2750	0.1730	0.2904	0.2250	0.4836	0.4054
1.5×10^7	0.5150	0.2950	0.4836	0.3637	0.7947	0.7006

For C-like languages, the transition from the non-optimized mode (-Od) to the optimized mode (-O2) demonstrates a performance boost of approximately 1.15x for C#, 1.3x for C++, and from 1.5x to 1.7x for C.

Go shows the greatest stability in performance as the problem size increases. Execution time increases almost proportionally to the increase in system size. Java and C# occupy an intermediate position, demonstrating fairly high efficiency due to Just-In-Time (JIT) compilation and advanced runtime optimization mechanisms.

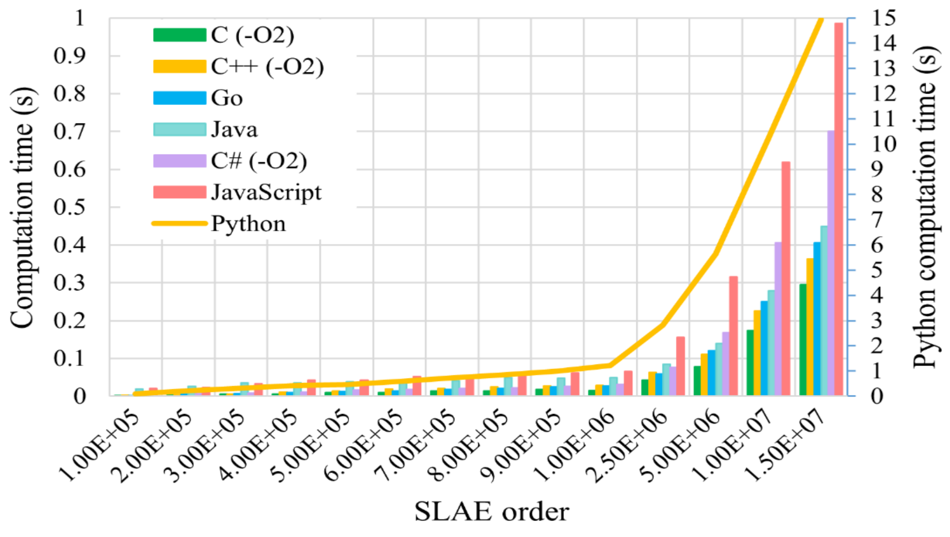

Fig. 1. General results of computational experiments

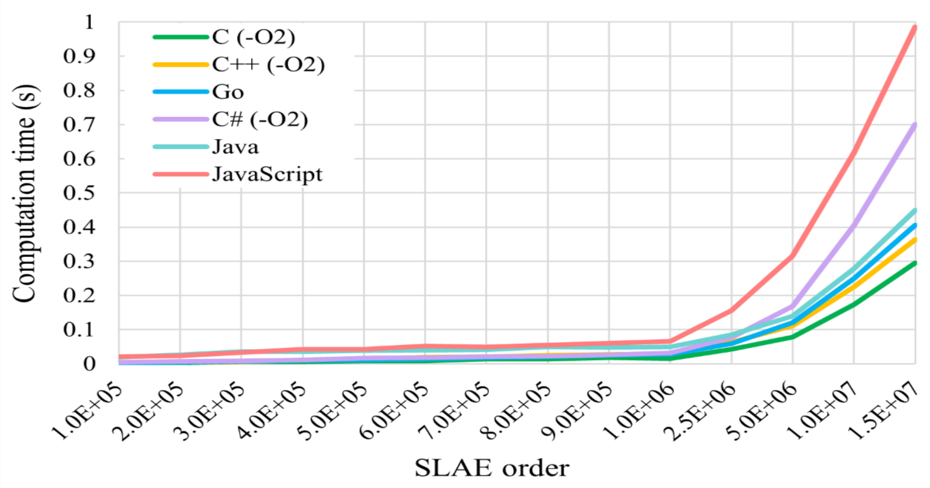

Fig. 2. Results of computational experiments for C-like languages, Java, JavaScript, and Go

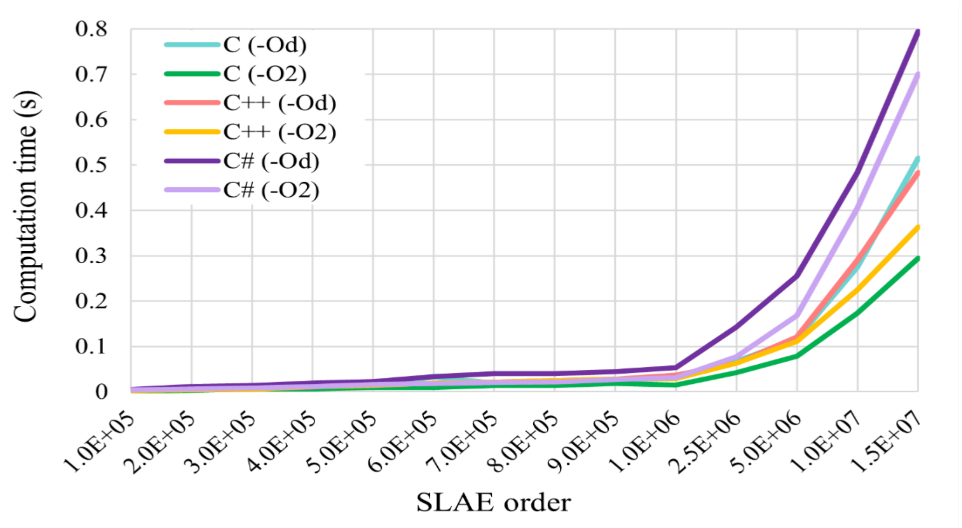

Fig. 3. Comparison of results in different optimization modes for C-like languages

The conducted experiments prove that for high-performance scientific computing, especially in computer simulation, compiled languages with explicit static typing, such as Go, C/C++, and Java, are the most effective, as they provide minimal overhead and high scalability of computational algorithms.

Conclusions

The conducted study presents a comprehensive analysis of the computational efficiency of modern programming languages in implementing numerical methods of mathematical physics, specifically the TDMA for solving SLAE with tridiagonal matrices.

The scientific novelty of this work lies in a systematic comparative study of the performance of various programming languages using a unified methodology for experimental research. This approach enables an objective quantitative assessment of the computational performance of modern languages in solving typical numerical mathematics problems.

Compiled languages, particularly C/C++, Go, and Java, demonstrated the highest performance and the lowest sensitivity to increasing problem size. In contrast, interpreted languages such as Python and JavaScript exhibited significantly lower computational efficiency, characterized by an exponential increase in execution time as the problem size grows. However, their advantage lies in the simplicity of development and rapid prototyping.

The transition from an unoptimized compilation mode to a fully optimized mode results in a speedup of approximately 1.15x for C#, 1.3x for C++, and between 1.5x and 1.7x for C, confirming the necessity of proper compiler configuration to achieve optimal computational performance. Meanwhile, the Go programming language demonstrated the highest stability in performance when scaling computational tasks. Go’s uniqueness lies in its combination of high execution speed, a relatively simple syntax, and built-in concurrency mechanisms.

A comparative analysis of C and C++ confirmed that these languages remain the most efficient for low-level computations, ensuring minimal overhead and a close-to-hardware implementation of algorithms. This conclusion aligns with the findings of previous research [1, 8, 11].

Future research directions include an in-depth study of the impact of computer system architecture, compiler optimization mechanisms, and runtime environment characteristics on the performance of computational algorithms. Additionally, expanding the range of numerical methods and programming languages under investigation will provide further insights into computational efficiency across different paradigms.

References

1. J. P. L. Escola, U. B. d. Souza, L. d. C. Brito. Discrete Wavelet Transform in digital audio signal processing: A case study of programming languages performance analysis. *Comput. Elect. Eng.* 2022. Vol. 104. DOI: https://doi.org/10.1016/j.compeleceng.2022.108439

2. N. D. Katopodes. Basic Concepts. *Free-Surface Flow*. 2019. P. 2–79. DOI: https://doi.org/10.1016/b978-0-12-815485-4.00007-3

3. TIOBE Index. URL: https://www.tiobe.com/tiobe-index (date of access: 15.03.2025).

4. T. Tanadechopon, B. Kasemsontitum. Performance Evaluation of Programming Languages as API Services for Cloud Environments: A Comparative Study of PHP, Python, Node.js and Golang. *2023 7th Int. Conf. Inf. Technol (InCIT)*. Chiang Rai, 16–17 Nov., 2023. P. 17–21. DOI: https://doi.org/10.1109/incit60207.2023.10413079

5. J. L. Hennessy, D. A. Patterson. A new golden age for computer architecture. *Commun. ACM*. 2019. Vol. 62, No. 2. P. 48–60. DOI: https://doi.org/10.1145/3282307

6. H. K. Dhalla. A Performance Analysis of Native JSON Parsers in Java, Python, MS.NET Core, JavaScript, and PHP. *2020 16th Int. Conf. Netw. Service Manage. (CNSM)*, Izmir, 2–6 Nov. 2020. P. 1–5. DOI: https://doi.org/10.23919/cnsm50824.2020.9269101

7. I. I. Zhulkovska, O. O Zhulkovskyi, V. V. Bilio. Typizatsiia suchasnykh mov prohramuvannia: zbirnyk naukovykh prats DDTU. *Tekhnichni nauky*. 2017. Vol. 30, No. 1. P. 154–158.

8. H. Eichhorn, R. Angerl, J. L. Cano, F. McLean. A Comparative Study of Programming Languages for Next-Generation Astrodynamics Systems. *6th International Conference on Astrodynamics Tools and TechniquesAt*. Darmstadt, March 2016. URL:

https://www.researchgate.net/publication/298577453_A_Comparative_Study_of_Programming_Languages_for_Next-Generation_Astrodynamics_Systems

9. Y. K. Gupta, S. Kumari. A Study of Big Data Analytics using Apache Spark with Python and Scala. *3rd Int. Conf. Intell. Sustain. Syst. (ICISS)*, Thoothukudi, 3–5 Dec. 2020. P. 471–478. DOI: https://doi.org/10.1109/iciss49785.2020.9315863
10. H. K. Omar, A. K. Jumaa. Big Data Analysis Using Apache Spark MLlib and Hadoop HDFS with Scala and Java. *Kurdistan J. Appl. Res.* 2019. Vol. 4, No. 1. P. 7–14. DOI: https://doi.org/10.24017/science.2019.1.2
11. A. Kumar, M. Goswami. Performance comparison of instrument automation pipelines using different programming languages. *Scientific Rep.* 2023. Vol. 13, No. 1. DOI: https://doi.org/10.1038/s41598-023-45849-y
12. J. W. Sunarto, A. Quincy, F. S. Maheswari, Q. D. A. Hafizh, M. G. Tjandrasubrata, M. H. Widianto. A Systematic Review of WebAssembly VS Javascript Performance Comparison. *Int. Conf. Inf. Manage. Technol. (ICIMTech)*. Malang, 24–25 Aug. 2023. P. 241–246. DOI: https://doi.org/10.1109/icimtech59029.2023.10277917
13. V. M. Ionescu, F. M. Enescu. Investigating the performance of MicroPython and C on ESP32 and STM32 microcontrollers. *2020 IEEE 26th Int. Symp. Des. Technol. Electron. Packag. (SIITME)*. Pitesti, 21–24 Oct. 2020. P. 234–237. DOI: https://doi.org/10.1109/siitme50350.2020.9292199
14. I. Plauska, A. Liutkevičius, A. Janavičiūtė. Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller. *Electronics*. 2022. Vol. 12, No. 1. P. 143. DOI: https://doi.org/10.3390/electronics12010143
15. P. Dymora, A. Paszkiewicz. Performance Analysis of Selected Programming Languages in the Context of Supporting Decision-Making Processes for Industry 4.0. *Appl. Sci*. 2020. Vol. 10, No. 23. P. 8521. DOI: https://doi.org/10.3390/app10238521
16. J. Gmys, T. Carneiro, N. Melab, E.-G. Talbi, D. Tuyttens. A comparative study of high-productivity high-performance programming languages for parallel metaheuristics. *Swarm Evol. Computation*. 2020. Vol. 57. DOI: https://doi.org/10.1016/j.swevo.2020.100720
17. D. Beronic, L. Modric, B. Mihaljevic, A. Radovan. Comparison of Structured Concurrency Constructs in Java and Kotlin – Virtual Threads and Coroutines. *2022 45th Jubilee Int. Conv. Inf., Communication Electron. Technol. (MIPRO)*. Opatija, 23–27 May 2022. P. 1466–1471. DOI: https://doi.org/10.23919/mipro55190.2022.9803765
18. O. O. Zhulkovskyi, I. I. Zhulkovska, V. V. Shevchenko. Evaluating the effectiveness of the implementation of computational algorithms using the OpenMP standard for parallelizing programs, *Inform. math. methods simul*. 2021. Vol. 11, No. 4. P. 268–277. DOI: https://doi.org/10.15276/imms.v11.no4.268
19. O. Zhulkovskyi, I. Zhulkovska, P. Kurliak, O. Sadovoi, Y. Ulianovska, H. Vokhmianin. Using asynchronous programming to improve computer simulation performance in energy systems. *Energetika*. 2025. Vol. 71, No. 1. P. 23–33. DOI: https://doi.org/10.6001/energetika.2025.71.1.2
20. C\C++ Documentation. URL: https://learn.microsoft.com/en-us/cpp (date of access: 15.03.2025).
21. C# and .NET Documentation. URL: https://learn.microsoft.com/en-us/dotnet (date of access: 15.03.2025).
22. Java Documentation. URL: https://docs.oracle.com/en/java (date of access: 15.03.2025).
23. JavaScript: Node.js Documentation. URL: https://nodejs.org/docs/latest/api (date of access: 15.03.2025).
24. Python 3 Documentation. URL: https://docs.python.org/3 (date of access: 15.03.2025).
25. Go Documentation. URL: https://go.dev/doc (date of access: 15.03.2025).

Oleg Zhulkovskyi Олег Жульковський	PhD, Associate Professor, Acting Head of the Department of Software Systems, Dniprovsky State Technical University https://orcid.org/0000-0003-0910-1150 e-mail: olalzh@ukr.net	Кандидат технічних наук, в.о. завідувача кафедри програмного забезпечення систем, Дніпровський державний технічний університет
Inna Zhulkovska Інна Жульковська	PhD, Associate Professor of Department of Cybersecurity and Information Technologies, University of Customs and Finance https://orcid.org/0000-0002-6462-4299 e-mail: inivzh@gmail.com	Кандидат технічних наук, доцент кафедри кібербезпеки та інформаційних технологій, Університет митної справи та фінансів, м.Дніпро
Hlib Vokhmianin Гліб Вохмянін	Master Student of the Department of Software Systems, Dniprovsky State Technical University https://orcid.org/0000-0002-9582-5990 e-mail: vohmyanin.yleb@gmail.com	Здобувач вищої освіти другого (магістерського) рівня, кафедра програмного забезпечення систем, Дніпровський державний технічний університет
Anastasiia Tkach Анастасія Ткач	Student of the Department of Software Systems, Dniprovsky State Technical University https://orcid.org/0009-0002-7784-0684 e-mail: anastasiatkach920@gmail.com	Здобувач вищої освіти першого (бакалаврського) рівня, кафедра програмного забезпечення систем, Дніпровський державний технічний університет