Analysis and optimization of quantitative characteristics of the formation of multi-threaded parallelism processes in BigData sorting tasks

Volodymyr Khilenko
Institute of Computer Engineering and Applied Informatics Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Bratislava, Slovakia
0000-0003-3491-8621

Oleksii Stepanov
Department of Information Technologies
National University of Life and Environmental Sciences of Ukraine
Kyiv, Ukraine
0000-0002-0939-6991

Dmytro Safonchyk
Department of Information Technologies
National University of Life and Environmental Sciences of Ukraine
Kyiv, Ukraine

Vasyl Khylenko Jr.
Institute of Computer Engineering and Applied Informatics Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Bratislava, Slovakia

Vladyslav Hlushchenko
Department of Information Technologies
National University of Life and Environmental Sciences of Ukraine
Kyiv, Ukraine

***Abstract*— A set of software and hardware totality for solving the problem of increasing the speed of processing big data through parallelization by organizing multi-threading is considered and analyzed. The solution of this problem is carried out in the aspect of the problem of increasing the productivity of processing big data for artificial intelligence and machine learning algorithms. Numerical modeling and analysis of processor time consumption when organizing multithreading were performed on examples of array sorting for matrices of different dimensions and trends in changes in the optimal number of threads were analyzed. The hypothesis of a monotonic (quasi-monotonic) dependence of the reduction of processor operating time with an increase in the number of threads was numerically verified. A comparative analysis of the implementation of parallel computing was carried out on hardware of different configurations and using different programming languages.**

Keywords*— *parallel computing, BigData, hardware and software systems, multithreading, matrix, bubble sort

I. Introduction. Research Issues and Problem Statement

The need for high-performance computing is constantly growing due to the increasing volumes of information that require processing. The ever-wider application of machine learning (ML) algorithms, artificial intelligence (AI), and smart control systems requires the implementation of methods and technologies to accelerate Big Data processing.

An effective direction in the development of software for high-performance computing remains the use of methods and algorithms for parallel data processing, as well as the parallelization of computing operations [1-3]. The increasing spread of distributed computing, information and control systems forms the hardware base and architectural solutions for the implementation of appropriate software tools that implement parallelism. An important area of industrial application of parallel computing is analytical algorithms for analysis and forecasting, which are a key element of automatic control systems and decision support systems for technical objects and technological processes [4-6]. The operation of these systems can be deployed on a hardware component that may consist of household computing devices such as laptops or desktop PCs, which are much less powerful than the servers of computing centers, but whose power may be sufficient for computing. For practical use in such systems, assessments of the effectiveness of the application of parallelism on existing hardware solutions and architectures are critical. Research on comparative data when using different parallelism schemes will allow choosing solutions that make it possible to minimize the time required for analysis and forecasting of the dynamics of an object or technological process.

The purpose of this work is to test the hypothesis of a monotonic (quasi-monotonic) dependence of the reduction in processor runtime with an increase in the number of threads on hardware of different configurations with an increase in the dimensionality of the processed information arrays. The results

979-8-3315-3562-9/25/$31.00 ©2025 IEEE

of this study should allow optimizing the organization of computing processes when working with Big Data and more accurately determining the optimal choice of hardware solutions, both in terms of the number of processor cores involved and the number of threads. The optimal selection of these indicators, combined with the correct choice of algorithmic support [7], should ensure the minimum time to solve the problem for hardware of a specific architecture and a given number of cores when using different algorithms for clustering the output information arrays.

II. Methodology of Research and Experimentation

We will consider the organization of multi-threaded data processing concerning information arrays in which it is necessary to perform element sorting. Solving this range of tasks involves performing arithmetic operations, specifically the operations executed in the bubble sort algorithm. The choice of these task classes is determined by their use in AI and ML algorithms when it is necessary to perform computational operations during the comparison of values of individual elements within an information array.

The algorithm for organizing parallel computations for the considered class of tasks when working with Big Data will include the following stages:

1. Clustering of the input information array (formation of a set of first-level clusters);

2. Comparison of elements within clusters and rearrangement of elements (sorting) in each cluster according to a defined rule, using multi-threading;

3. Formation of an updated, from the perspective of clustering, information array using the results of item 2.

A new set of clusters (second-level clusters) is formed by combining elements with similar values, selected from the set of first-level clusters. The mathematical evaluation, based on which elements from the first-level clusters are combined into new clusters (second-level clusters), is defined by the user. Such a two-level clustering technology can be useful and applied in many practical tasks related to the application of AI and ML algorithms [8-11].

The model experiments and calculations of this work are focused on finding optimal solutions for performing the second stage. Operations involving the rearrangement of elements in separate fragments of the array in descending (ascending) order can be performed in parallel – both for separate clusters and within each cluster.

Model computational experiments were conducted for matrices with dimensions of 200×200, 1000×1000 and 5000×5000.

The calculations and time measurement were programmed according to the following algorithm:

1. Create an array of a given dimension n×n with random numbers.

2. Set $t = t_{01}$ and start counting time.

3. Sort the elements for the array rows. Sorting of each row occurs in one thread, one after the other, of all rows of the array.

4. Set the completion time of the operation t_F and determine $t_{F1} = t_F - t_{01}$.

5. Restore the original array.

6. Set the time $t = t_{02}$.

7. Sort in two threads: the first thread sorts the first row, the second thread sorts the second row, and whichever thread finishes faster sorts the third row, and the other thread sorts the fourth row, and so on until all rows of the array are sorted.

8. Fix the completion time of the operation t_F and determine $t_{F2} = t_F - t_{02}$.

9. Restore the original array.

10. Fix the time $t = t_{03}$.

11. Sort in three threads: the first thread sorts the first row, the second - the second row, the third - the third row. Whichever one finishes faster - sorts the fourth row. The next free thread sorts the fifth row, another free thread sorts the sixth row, and so on, until all rows are sorted.

12. Fix the completion time of the operation t_F and determine $t_{F3} = t_F - t_{03}$.

13. Repeat points 9,10,12 using the row distribution method as specified in point 11 up to 100 threads

14. Repeat points 1-13 2 more times for all sorting options again from 1 to 100 threads.

15. Calculate the average time for each option for the number of threads.

III. Modeling Results and Analysis

Results of multithreaded computation of sorting operation execution time for matrices of size 200×200, 1000×1000 and 5000×5000 are presented in Fig. 1-9. The calculations were performed using different programming languages and on hardware with different characteristics.

Results of calculating the dependence of the sorting task execution time are presented:

- when writing software in Python - in Fig. 1-3;

- when writing software in C++ in Fig. 4-6;

- when writing software in Java - in Fig. 7-9.

Characteristics of devices and programming languages used in the study are shown in Table 1. Devices with similar configurations were used in the tests, which made it possible to investigate their features on the speed of calculations. The following main parameters were selected for analysis:

- operating system type and version;

- processor model, which characterizes the base and maximum frequency, number of cores and logical processors, and size of cache memory of different levels;

- amount of RAM.

TABLE I. CHARACTERISTICS OF DEVICES AND PROGRAMMING LANGUAGES USED IN THE STUDY

№	Programming languages	Operating system	Processor model	Base frequency/ maximum frequency	Number of cores/ logical processors	L1 cache per core/ L2 cache per core/ L3 cache total	RAM size
1	C++ Java Python	Windows 11 Pro, version 24H2	AMD Ryzen 7 4800H	2.9 GHz/ 4.2 GHz	8/16	64 KB/ 512 KB/ 8 MB	32,0 GB, DDR4-3200MHz
2	C++ Java Python	Windows 11 Home version 23H2	AMD Ryzen 7 4800HS	2.9 GHz/ 4.2 GHz	8/16	64 KB/ 512 KB/ 8 MB	16,0 GB, DDR4-3200 MHz
3	C++ Java Python	Windows 11 Pro version 24H2	Intel Core i5-11400H,	2.7 GHz/ 4.5 GHz	6/12	96 KB/ 1.25 KB/ 12 MB	16,0 GB DDR4-3200 MHz
4	C++	Windows 10 Pro version 22H2	AMD Ryzen 5 3600	3.6GHz/ 4.2 GHz	6/12	96 KB/ 512 KB/ 32 MB	32,0 GB, DDR4-3600 MHz
5	Java	Windows 11 version 23H3	AMD Ryzen 7 5800H	3.20 GHz 4.4 GHz	8/16	64 KB/ 512 KB/ 16 MB	16,0 GB, DDR4-3200 MHz

Next the short analysis of the obtained calculation results is conducted

A. Testing Results for the Python Language

Although the use of the most common implementations of the Python language is not recommended for developing high-performance parallel computing applications due to the presence of the Global Interpreter Lock (GIL) [12], we conducted testing of the given task using this language. The code was implemented without the use of external libraries, such as NumPy or SciPy, and the obtained results are shown in Fig. 1-3.

To work around GIL limitations, the multiprocessing module was used via the *concurrent.futures.ProcessPoolExecutor* interface. The number of processes was limited by setting *max_processes = os.cpu_count() * 2*, since using a greater number of processes on computers with the aforementioned characteristics began to produce unstable results. As can be seen, all graphs are quite similar and differ only slightly depending on the computer's specifications.

The data presented in Fig. 1 show that in the initial stage, increasing the number of threads for a 200×200 matrix results in only a slight reduction in execution time across all hardware types. Specifically, for the first device, the minimum time is 0.70 s. using 3 processes; for the second, 0.69 s. with 2 processes; and for the third, 0.57 s. with 3 processes. It can also be observed that after a certain point, the computation time becomes longer than that in single-threaded mode – that is, when only one processor core is used to sort the entire matrix. This occurs at 7 processes for the first device, 4 processes for the second, and 6 processes for the third. This indicates that more time is spent organizing parallel computation than performing the calculation itself, and as the number of processes increases, the computation time grows almost linearly.

The increase in performance can be estimated by the ratio of the single-process computation time to the minimum observed time. For the first device, this value is 1.21; for the second, 1.02; and for the third, 1.17. The total computation time, calculated as the sum of the durations of all three test runs, is: 2 m. 45.54 s. for the first device, 3 m. 8.58 s. for the second, and 1 m. 43.32 s. for the third.

Fig. 2 shows the computation results for a 1000×1000 matrix. As can be seen from the graph, the data for the third device initially closely align with the second device, and later with the first. The minimum execution time was 7.82 s. for the first device, 7.53 s. for the second, and 8.17 s. for the third, all using 8 processes. A gradual and slight increase in computation time is then observed. At the final point – using 32 processes for the first and second devices and 24 for the third – the difference from their respective minimum times is 2.62 s. for the first device, 3.97 s. for the second, and 1.85 s. for the third. This indicates that the computation time for the second device increases more rapidly.

The performance gain is 6.39 for the first device, 4.36 for the second, and 4.06 for the third. The total computation time was: 18 m. 27.53 s. for the first device, 17 m. 34.84 s. for the second, and 12 m. 32.76 s. for the third.

In the next test, which was conducted only on the first and second devices and is shown in Fig. 3, the results for a 5000×5000 matrix are presented. The minimum execution time was 962.41 s. on the first device using 9 processes, and 885.27 s. on the second using 8 processes. As in Fig.2, a slight increase in execution time is observed. At the final point, with 32 processes, the increase from the minimum is 43.86 s. for the first device and 375.40 s. for the second. Again, the second device shows a faster increase in execution time.

Performance gain for the first device is 6.55; for the second, 4.86. Total computation time: 35 h. 4 m. 2.12 s. for the first device, and 35 h. 31 m. 23.45 s. for the second.

The presented results indicate that the time to reach the minimum execution point correlates with hardware parameters – particularly the amount of RAM.

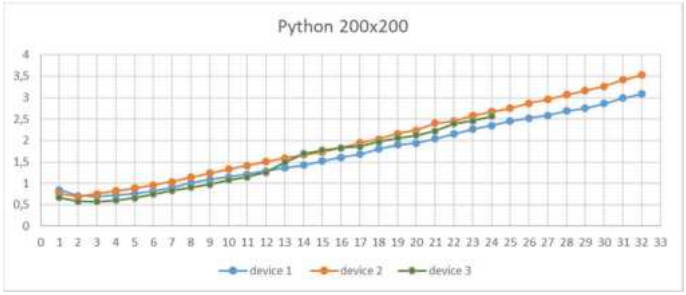

Fig. 1. Testing results for a 200×200 matrix using Python

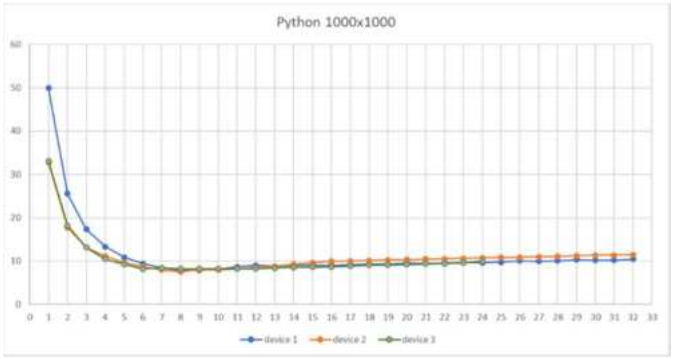

Fig. 2. Testing results for a 1000×1000 matrix using Python

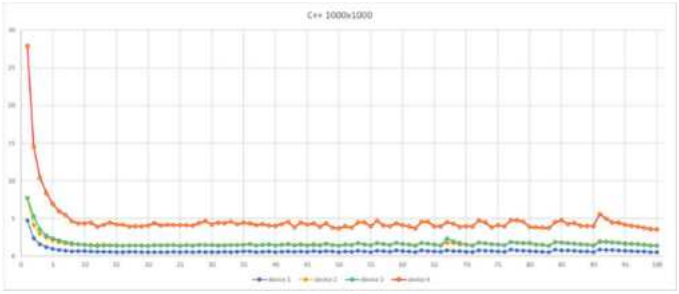

Fig. 3. Testing results for a 1000×1000 matrix using C++

B. Testing Results for the C++ Language

The trend of graph similarity is also observed for C++, as shown in Fig. 4-6. For matrices sized 200×200 and 1000×1000, four devices were used.

The testing results for a 200×200 matrix are presented in Fig. 4. For the first device, the minimum time is 4.80 ms. with 15 processes; for the second device, 22.77 ms. with 11 processes, with the next closest time being 22.84 ms. using 14 processes; for the third device, 15.26 ms. with 10 processes; and for the fourth device, 30.49 ms. with 50 processes. Notably, a time very close to the minimum – 30.96 ms. – was recorded with 11 processes.

The performance speedup is 6.49 for the first device, 3.01 for the second, 4.04 for the third, and 7.45 for the fourth. The total time spent on computation is 2.54 s. for the first device, 15.77 s. for the second, 8.93 s. for the third, and 15.92 s. for the fourth. For the second device, starting from 67 processes, the computation time exceeds the execution time in single-threaded mode.

Fig. 4. Testing results for a 200×200 matrix using C++

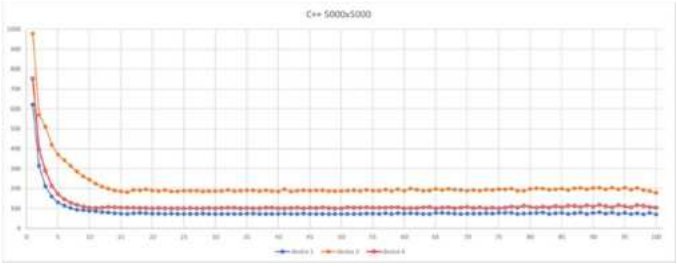

Fig. 5. Testing results for a 5000×5000 matrix using C++

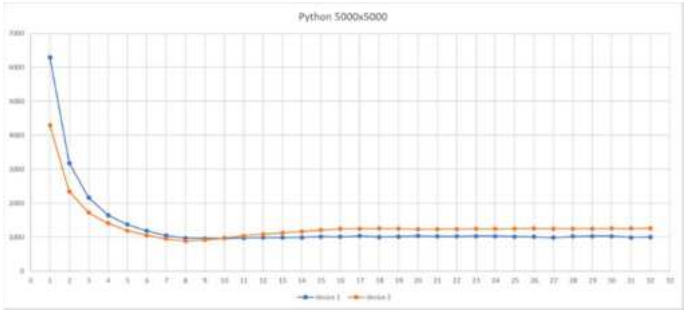

Fig. 6. Testing results for a 5000×5000 matrix using Python

An interesting result of the study is that almost all devices exhibit a sharp increase in computation time at the same step sizes. However, there is no clear correlation between the minimum time and the number of cores or logical processors.

Fig. 5 shows the testing results for a 1000×1000 matrix. For the first device, the minimum time is 0.52 s. with 20 processes; the next closest previous time is 0.54 s. with 16 processes. For the second device, the minimum is 1.385 s. with 20 processes; the previous closest time is 1.391 s. with 16 processes. For the third device, the minimum is 1.35 s. with 12 processes, and for the fourth device, 3.53 s. with 100 processes, although a close-to-minimum time of 3.93 s. was recorded with 12 processes.

The performance speedup is 9.0 for the first device, 5.59 for the second, 5.67 for the third, and 7.89 for the fourth. The total computation time is 3 m. 34.89 s. for the first device, 8 m. 29.86 s. for the second, 8 m. 31.95 s. for the third, and 23 m. 34.18 s. for the fourth. Here again, nearly all devices show a simultaneous increase in computation time at the same step intervals. However, unlike the previous case, this time there is a clear match between the minimum time and the number of logical processors – or at least with a value very close to the minimum.

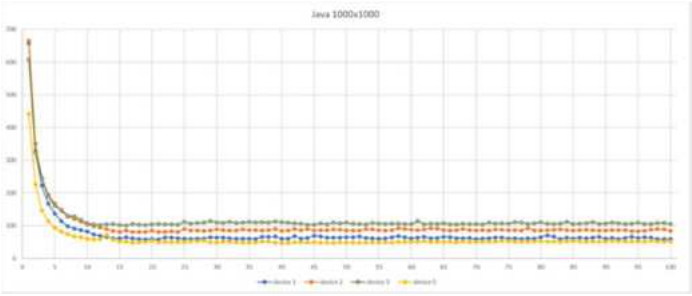

Fig. 7. Testing results for a 1000×1000 matrix using Java

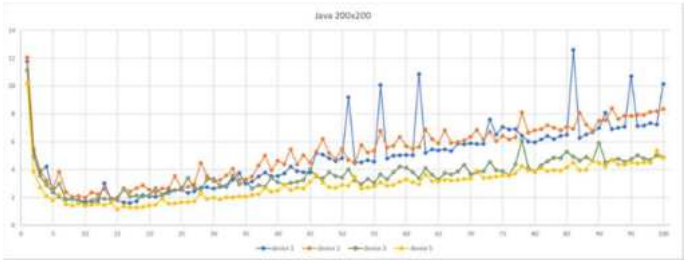

Fig. 8. Testing results for a 200×200 matrix using Java

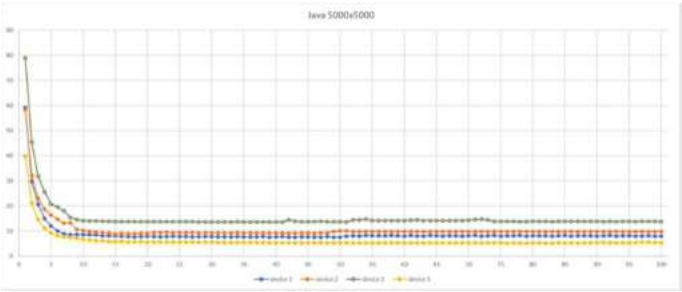

Fig. 9. Testing results for a 5000×5000 matrix using Java

The experiment shown in Fig. 6, involving a 5000×5000 matrix, was conducted without the third device. The results for the other devices are as follows: for the first device, the minimum time is 71.7 s. with 100 processes, although a close time of 72.31 s. was recorded with 16 processes; for the second device, 178.9 s. with 100 processes, with a similar result of 182.95 s. at 16 processes; for the fourth device, 100.532 s. with 50 processes, while a close value of 102.68 s. was seen at 11 processes. However, at 12 processes, the time rose to 105.09 s. – a significant deviation from both the minimum and neighboring values.

The performance speedup is 8.66 for the first device, 5.47 for the second, and 7.49 for the fourth. The total computation time was 7 h. 13 m. 38.78 s. for the first device, 18 h. 4 m. 10.62 s. for the second, and 9 h. 55 m. 28.47 s. for the fourth.

C. Testing Results for the Java Language

Similar patterns are observed when using Java; these results are shown in Fig. 7-9. All devices from Table 1 except the fourth participated in this part of the study.

Fig. 7 presents the results for a 200×200 matrix. For the first device, the minimum time is 1.59 ms. with 17 processes; for the second, 1.76 ms. with 15 processes; for the third, 1.57 ms. with 10 processes; and for the fifth, 1.15 ms. with 15 processes.

The performance speedup is 7.39 for the first device, 6.76 for the second, 7.11 for the third, and 8.89 for the fifth. The total computation time is 1.47 s. for the first device, 1.56 s. for the second, 1.08 s. for the third, and 0.89 s. for the fifth.

Here as well, there is a noticeable spike in computation time at certain steps, with no clear alignment between the minimum time and the number of cores or logical processors.

Fig. 8 shows the results for a 1000×1000 matrix. For the first device, the minimum time is 58.167 ms. with 21 processes, with nearby values of 61.176 ms. at 17 processes and 64.56 ms. at 16 processes. For the second device, the minimum is 80.91 ms. at 21 processes, with 81.05 ms. at 17 processes and 85.14 ms. at 16. The third device's minimum is 100.61 ms. at 16 processes, with 102.4 ms. at 12. For the fifth device, the minimum is 46.76 ms. at 41 processes, with a close time of 48.60 ms. at 17 processes and 51.36 ms. at 16.

The performance speedup is 11.29 for the first device, 8.22 for the second, 6.05 for the third, and 9.45 for the fifth. Total computation time is 22.93 s. for the first device, 30.13 s. for the second, 35.34 s. for the third, and 17.91 s. for the fifth.

Devices 1, 2, and 5 show no clear match between the minimum or near-minimum time and the number of logical processors. Time spikes occur at certain steps, but no strong correlations are observed.

Fig. 9 shows testing results with a 5000×5000 matrix in Java. For the first device, the minimum time is 7.49 s. with 50 processes, with a near-minimum of 7.69 s. at 18 processes and 7.91 s. at 16. For the second, the minimum is 8.81 s. at 15 processes and 9.04 s. at 16. The third device shows 13.58 s. at 38 processes, with 13.71 s. at 17 and 13.96 s. at 12. The fifth device has a minimum of 5.19 s. at 83 processes, and a nearby value of 5.52 s. at 23 processes. At 16 processes, it records 5.77 s. – a significant deviation.

Performance speedup is 7.88 for the first device, 6.62 for the second, 5.81 for the third, and 7.66 for the fifth. Total computation time is 44 m. 45.7 s. for the first device, 53 m. 25.65 s. for the second, 1 h. 16 m. 37.61 s. for the third, and 30 m. 49.75 s. for the fifth.

For all devices, no clear match is observed between minimum time and the number of logical processors. However, time spikes are generally seen when using around 50 processes.

A detailed analysis of the curves in Fig. 1-9 shows that there are synchronous increases in computation time at some points. With certain assumptions, there are also cases where the minimum time aligns with the number of cores or logical processors. It can be noted that the best performance and the lowest computation time are achieved not only when using processors with a higher number of cores but also with a larger amount of RAM. This difference is especially noticeable when computing large matrices. For example, when comparing devices 1 and 2 while computing a 5000×5000 matrix using C++, the greater RAM size accelerated computation by 2.5 times. The operating system likely also influences how parallel computations are organized. However, since the tests

were conducted using the same OS (albeit different versions), no significant impact from version differences was observed.

When comparing CPUs by manufacturer and release year, only minor variations were noted, assuming the CPUs were of similar age. Nonetheless, newer processors naturally delivered lower computation times. Cache size (either of cores or the CPU) had no significant effect on performance based on observed results.

A substantial difference was seen across programming languages: Java programs consistently demonstrated the fastest performance. Regarding the optimal number of processes, for Python the best performance typically aligns with the number of physical cores, while for C++ and Java, the number of logical processors serves as a good reference. However, there were cases where using the number of processes equal to logical cores resulted in notable deviations from the minimum computation time. Although the performance gains were substantial, they were not always optimal.

Another common characteristic for the C++ and Java languages with matrix dimensions of 1000×1000 and 5000×5000 is that with an increase in the number of processes, the calculation time does not always increase, and values close to the minimum can be obtained when using a number of processes that is several times larger than the number of cores. Sometimes, some periodicity in the appearance of almost minimum calculation time can be recorded.

IV. Conclusion

The computational experiments on multithreading for matrices of various sizes showed that a strictly monotonic decrease in execution time with an increasing number of threads was not observed.

The minimum execution time is achieved at a certain number of threads, k_{opt} which is optimal in terms of minimizing execution time. For this optimal number of threads, the following relation holds:

$$t_{com} = t_{min}$$

or

$$t_{com} = \varepsilon\left(t_{min}\right)$$

where, t_{com} is the actual processor execution time, t_{min} is the minimum achievable execution time, ε (t_{min}) is a user-defined tolerance for insignificant deviations from t_{min}.

On the graphs presented, k_{opt} represents the “inflection point” of the curve.

Further increasing the number of threads, depending on the matrix size, either leads to t_{com} stabilizing around t_{min} or a (quasi-)monotonic increase in t_{min}.

For small-sized arrays, k_{opt} usually correlates with the number of processor cores.

1. The execution time of operations is correlated with the number of processor cores, however, as in the case of changing threads, this dependence is nonlinear.

2. Increasing the dimensionality of matrices (in the considered range) does not lead to violations of the established dependencies of the change in the optimal time for solving a problem using multi-threaded parallel computing.

3. The conclusions obtained relate to a specific class of performed operations, in particular, sorting operations in hierarchical clustering technology, which are quite common in AI and ML algorithms. For this class, the presented results can be useful when making decisions on optimizing the parameters of computational procedures using parallelism. In general, the effectiveness of organizing parallel computing correlates with the class of problems being solved, therefore the final choice of parallelization solutions should take into account the specifics of a particular problem.

References

[1] Z.Li, S.Yuan, Z.Guan. Robust and Scalable Federated Learning Framework for Client Data Heterogeneity Based on Optimal Clustering. (2025) Journal of Parallel and Distributed Computing, Volume 195, art. no. 104990. DOI: 10.1016/j.jpdc.2024.104990

[2] V.V. Khilenko. Organization of parallel computational processes to compute step boundary layer models. (1998) Cybernetics and Systems Analysis, 34 (2), pp. 305 - 308, DOI: 10.1007/BF02742083

[3] X.Chen, S.Hu, C.Yu, Z.Chen, G.Min. Real-Time Offloading for Dependent and Parallel Tasks in Cloud-Edge Environments Using Deep Reinforcement Learning. (2024) IEEE Transactions on Parallel and Distributed Systems, 35 (3), pp. 391 – 404. DOI: 10.1109/TPDS.2023.3349177

[4] A.La Bella, L.Nigro, R.Scattolini. Predictive Control and Benefit Sharing in Multi-Energy Systems. (2024) IEEE Transactions on Control Systems Technology, 32 (2), pp. 368 – 383. DOI: 10.1109/TCST.2023.3310891

[5] V.V.Khilenko. Convergence of algorithms of the order reduction method in analysis of nonlinear mathematical models of fragments of automatic control systems and process control systems. (2001) Cybernetics and Systems Analysis, 37 (3), pp. 373 – 380. DOI: 10.1023/A:1011989710902

[6] T.Sun, K.L.Teo, X.-M.Sun. Numerical optimal control for switched nonlinear systems with inequality path constraints. (2023) Systems and Control Letters, 182, art. no. 105653. DOI: 10.1016/j.sysconle.2023.105653

[7] V.V.Khilenko, O.V.Stepanov, I.Kotuliak, M.Reis. Optimization of the Selection of Software Elements in Control Systems with Significantly Different-Speed Processes. (2021) Cybernetics and Systems Analysis, 57 (2), pp. 185 – 189. DOI: 10.1007/s10559-021-00342-0

[8] G.Nguyen, S.Dlugolinsky, V.Tran, Á.López García. Network security AIOps for online stream data monitoring. (2024) Neural Computing and Applications, 36 (24), pp. 14925 – 14949. DOI: 10.1007/s00521-024-09863-z

[9] J.-H.Hu, L.-W.Kang. P.-C.Chang. PADU-Net: Parallel Attention-based Dual U-Net for Retinal Vessel Segmentation. (2024) IEEE 13th Global Conference on Consumer Electronics (GCCE), Kitakyushu, Japan, 2024, pp. 152-153, DOI: 10.1109/GCCE62371.2024.10760672

[10] V.V.Khilenko. An Algorithm for Decomposition Control and Prediction of Trajectories of Nonlinear Stochastic Systems Under Different-Speed Processes in Their Dynamics. (2022) Cybernetics and Systems Analysis, 58 (2), pp. 220 – 224. DOI: 10.1007/s10559-022-00453-2

[11] L.-W.Kang, C.-C.Hsu, I.-S.Wang, T.-L.Liu, S.-Y.Chen, C.-Y.Chang. Vehicle trajectory prediction based on social generative adversarial network for self-driving car applications. (2020) Proceedings - 2020 International Symposium on Computer, Consumer and Control, IS3C 2020, art. no. 9394191, pp. 489 – 492. DOI: 10.1109/IS3C50286.2020.00133

[12] G.Zaccone. Python Parallel Programming Cookbook – Second Edition. 2019. Publisher: PackT Publishing. ISBN: 9781789533736.