UDC 004.42
DOI https://doi.org/10.32782/IT/2022-3-4

Valerii PAVLOV
PhD, Associate Professor, Associate Professor at the Department of Computer Engineering, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Peremohy Avenue, 37, Kyiv, Ukraine, 03056, pavlovvg@ukr.net
ORCID: *0000-0002-4299-0319*

To cite this article: Pavlov V. (2022). Ways to achieve internal parallelism of tasks in multithreaded computing. *Information Technology: Computer Science, Software Engineering and Cyber Security,* 3, 27–33, doi: https://doi.org/10.32782/IT/2022-3-4

WAYS TO ACHIEVE INTERNAL PARALLELISM OF TASKS IN MULTITHREADED COMPUTING

The article analyzes the evolution of the architecture of computer systems and identifies a way to improve the performance of calculations. It consists in the use of multi-threaded technologies as a set of hardware, software and methodological solutions. Multi-core processors are considered as the object of research. Each of the cores has its own cache memory, and at this level they can be considered as computing devices with distributed memory. At the same time, with large amounts of computation, this memory is not enough, so the shared memory is used, that is, they are transformed into devices with a common memory. Therefore, in terms of memory access, multi-core processors can be considered as hybrid computing devices. The goal of the work is to analyze approaches to optimizing transformations of calculations in the form of iterative cycles with subsequent experimental verification of their effectiveness on computers with multi-core processors. The novelty of the approach lies in the fact that traditionally arithmetic cycles are considered in optimization problems, which is a more particular problem. The role of the compiler is highlighted in code optimization by searching for internal parallelism. Attention is paid to cycles as the most promising object for parallelization. The indicator of «Depth of Inter-Step Communication in the Loop» (DISCL) is proposed, and the conditions for parallelization of cycles are formulated. The problem of load balancing during parallelization is considered. The results of experiments on the use of MPI technology for parallelization of some computational tasks, which use iterative cycles (sequences and numerical integration), are presented. As a criterion for the performance of calculations, invariant to the characteristics of various computer systems, the indicator λ is proposed. It designates the relative part of parallel computing in their total volume.

Key words: compute performance; paralleling compiler; internal parallelism; circle optimization.

Валерій ПАВЛОВ
кандидат технічних наук, доцент, доцент кафедри обчислювальної техніки, Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», просп. Перемоги, 37, Київ, Україна, 03056, pavlovvg@ukr.net
ORCID: *0000-0002-4299-0319*

Бібліографічний опис статті: Павлов В. (2022). Шляхи досягнення внутрішнього паралелізму задач при багатопотікових обчисленнях. *Information Technology: Computer Science, Software Engineering and Cyber Security,* 3, 27–33, doi: https://doi.org/10.32782/IT/2022-3-4

ШЛЯХИ ДОСЯГНЕННЯ ВНУТРІШНЬОГО ПАРАЛЕЛІЗМУ ЗАДАЧ ПРИ БАГАТОПОТІКОВИХ ОБЧИСЛЕННЯХ

У статті проаналізована еволюція архітектури комп'ютерних систем и визначений шлях підвищення продуктивності обчислень. Він полягає у використанні технологій багатопотіковості., як сукупності апаратних, програмних та методологічних рішень. У якості об'єкту дослідження розглядаються багатоядерні процесори. Кожне з ядер має власну кеш-пам'ять, і на цьому рівні вони можуть розглядатися, як обчислювальні пристрої з розподіленою пам'яттю. У той же час, при великих обсягах обчислень цієї пам'яті недостатньо, тому використовується загальна пам'ять, тобто відбувається їх трансформація у пристрої зі загальною пам'яттю. Тому багатоядерні процесори по доступу до пам'яті можуть розглядатися як гібридні обчислювальні пристрої.

Метою роботи є аналіз підходів до оптимізуючих перетворень обчислень у вигляді ітераційних циклів з наступною експериментальною перевіркою їх ефективності на комп'ютерах з багатоядерними процесорами. Новизна підходу заключна у тому, що традиційно у задачах оптимізації розглядаються арифметичні *цикли, що є більш строщеною задачею. Визначена роль компілятору в оптимізації коду за рахунок пошуку внутрішнього паралелізму. Приділено увагу циклам, як найбільш перспективному об'єкту для розпаралелювання. Запропонований показник «глибіни міжкрокового зв'язку у циклі» (ГМКЦ) та сформульовані умови для розпаралелювання циклів. Розглянута проблема балансування навантаження при розпаралелюванні. Надані експериментальні результати по застосуванню технології MPI для роспаралелювання деяких обчислювальних задач, що використовують ітераційні цикли (ряди и чисельне інтегрування). У якості критерію продуктивності обчислень, інваріантного до характеристик різноманітних комп'ютерних систем запропонований показник λ, що характеризує відносну частку паралельних обчислень у їх загальному обсязі.*

Ключові слова: *продуктивність обчислень; компілятор, що розпаралелює; внутрішній паралелізм; оптимізація циклів.*

Urgency of the problem. For more than 50 years, industrial progress has been determined by the level of development of computer equipment and technologies. With the increasing amount of information, higher and higher requirements for the performance of computer systems are put forward («Moore's Law» – Gordon Earle Moor, 1965). Until recently, this was achieved through the development of the technical component, namely, increasing the speed of information processing in various components of computer systems. However, today the potential of this path, for known reasons (leakage currents, power consumption and heat sink), has almost been exhausted and has reached its maximum. Therefore, technologies related to the simultaneous processing of information by parallel devices come to the fore. First, parallel memory banks began to be used, and then multiprocessor systems.

Analysis of recent research and publications. A number of publications on the topic of high-performance computing focus on the architecture of computing systems and their hardware component. According to well-known Flynn's taxonomy (Flynn Michel. J. 1966), multiprocessor computer systems belong to the **MIMD (Multiple Instruction, Multiple Data)** class. The issues of implementation of supercomputers based on cluster systems are most dynamically covered in periodically published reviews (Internet-resources: www.top500.org; www.nvidia.com; www.itc.ua/ua/; www.overclockers.ua/ etc.).

The idea of creating multi-core processors, as an evolution of the SMP architecture, is based on a nonlinear relationship between three main characteristics:

– clock frequency;
– power consumption and heat sink;
– compute performance.

Since the bulk of the power consumption in the processor is dynamic power $\boldsymbol{P_d}$, which is spent on switching CMOS switches and charging capacitors, it can be determined by the formula (W. Wolf, 2002):

$$P_d = CV^2 f \tag{1}$$

In turn, the switching time is inversely proportional to the applied voltage, therefore, the switching frequency $\boldsymbol{f}$ is directly proportional to the voltage **V**. This makes it possible to assert that the power consumption of the processor $\boldsymbol{P_d}$ is proportional to the third degree of frequency $\boldsymbol{f}$.

Empirically, it is established that when the clock frequency is increased by **20%**, the processor performance increases by only **13%**. This phenomenon is called the «The Moore's gap». At the same time, power consumption will increase by $\mathbf{(1.2)^3}$ times, that is, by **73%**!

Such a nonlinear dependence justifies the feasibility of replacing a single-core processor with a multi-core processor with the same power consumption, however, with greater total performance.

The second part of the publications is devoted to the issues of parallel computing software.

A kind of «basic» programming languages, the development of which has provided support for parallel computing, are **FORTRAN** and **C/C++**.

However, there are several basic standards for building parallel computing processes:

POSIX Threads – execution thread implementation standard;

OpenMP – parallel programming standard in a shared memory environment (**SMP**-systems) (Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, Jeff McDonald, 2000);

MPI – a standard for the transfer of messages between parallel executable processes in computing systems with distributed memory, for example, massively parallel systems, clusters and **GRID** systems (William Gropp, Ewing Lusk, Anthony Skjellum, 1999).

The latter two technologies have long been dominant in the field of parallel computing, so let's look at them in a little more detail.

There are their implementations for various programming languages, which allows when developing a program to achieve the necessary independence from the features of the computer system on which it will be executed.

Finally, it is possible to distinguish a large group of publications that consider the algorithms for building parallel calculations (V.V. Voevodin, 2002; V.P. Gergel, 2009; E.N. Gordeev, 2011). However, only arithmetic cycles are considered as the object of optimization, while most numerical methods are based on iterative cycles. In cycles of this type, the number of repetitions is not known in advance, so it is impossible to build a tier-parallel form of the calculation algorithm in advance and decompose the data.

The ultimate goal of using any parallel programming technology is to get the most productive program. However, these technologies themselves are essentially a set of directives that are taken into account in one way or another at the compilation stage. Therefore, it is the compiler that ultimately determines the result in the form of a machine program. There are techniques that allow to improve the properties of grammars even at the stage of recognizing chains of characters and building a parse tree (V. Pavlov, 2016).

If the compiler cannot find snippets of programs that can run in parallel, then it will generate sequential code of the machine program in this case.

The main stages of compilation are presented in Fig.1:

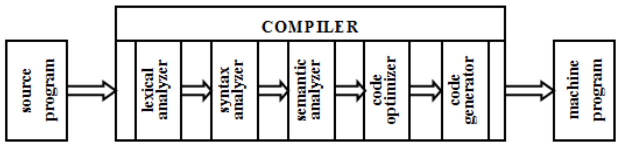


Fig. 1. Stages of a compiler

When lexical, syntactic, and semantic parsing is performed, code optimization is begun. The main goal of optimization is to get faster and smaller code.

Code optimization can be divided into global and local, where the latter affects only some blocks. Such blocks include, for example, linear sections of the program and cycles. The optimization process consists in the application of some techniques that should lead to the desired result – «optimizing transformations» (Aho Alfred V., Ullman Jeffrey D, 1979).

The goal of the article is to analyze approaches to optimizing transformations of calculations in the form of iterative cycles with subsequent experimental verification of their effectiveness on computers with multi-core processors.

Presentation of the main material of the study. A well-known proverb says that it is useless to look for a black cat in a dark room. To paraphrase, you can only parallelize those calculations for which it is possible to do so. That is, initially the algorithm for solving the problem should contain computation, which in principle can be performed simultaneously without compromising the result, that is, have internal parallelism (V.V. Voevodin, 2002). If it is missing it will not be possible to get the acceleration of calculations, even if there is an unlimited number of computing devices.

This is best illustrated by the well-known Amdahl's law (Amdahl Gene M., 1967), according to which the acceleration of calculations due to the use of several computing devices is possible only in that part of the program, where parallel computing is present. In other words, if the more of the computation in the program can be parallelized, then the acceleration is higher from the use of parallel computing devices.

However, in many cases, the initial parallelism of calculations is either hidden or absent altogether. To identify hidden parallelism in the algorithms, it is proposed to use the directed graph of the algorithm (V.V. Voevodin, 2002). In this case, the number of tiers is taken as a criterion for the effectiveness of this algorithm. The concept of unlimited parallelism was also formulated, which implies the presence of an unlimited number of processors with shared memory and instant transmission of messages between them.

Therefore, the main direction of improving the performance of calculations is to improve the algorithms for executing programs

Loop optimization. Loops as a group of repetitive calculations are inherent in almost all programming languages and most programs. They take a significant time, so their optimization can significantly increase the speed of calculations. These transformations include (Aho Alfred V., Sethi Ravi, Ullman Jeffrey D, 1986):

– Code motion beyond the loop;

– Removal of Induction Variables, which can be counted;

– Replacing Complex Operations with Simpler and Faster ones;

– Loop Unrolling;

– Loop fission (sub loops);

– Loop Inversion.

Obviously, the loop execution time is proportional to the number of loops and the execution time of a single step. Therefore, the ways to reduce loop execution time are either to reduce the number of loops or to reduce the duration of operations at each step.

To use parallel computing, the first method is preferable, because it guarantees uniform distribution and load balance on computing devices. If there is a loop of **N** repetitions of calculations, then it can be distributed to **P** computing devices, determining that each will perform **N/P** steps of the loop.

But this is not always possible if the results calculated at one step of the loop are used at subsequent steps in the loop (V.P. Gergel, V.A. Fursov, 2009). To assess this, we will use the concept of «Depth of Inter-Step Communication in the Loop» (**DISCL**), which will show the maximum number of preceding steps in the loop, the data from which is used in the current step. For example, when calculating the usual sum of a series, the **DISCL** is equivalent to **1**, since it is needed for calculating and adding to the sum the previous terms only. And when calculating the Fibonacci sequence, you need to know the results of the last two steps, so **DISCL** is **2**. The value of **DISCL** shows the relationship between the individual steps of the loop, the larger it is, and the more difficult it is to carry out calculations in parallel.

Ideally, of course, it is necessary that the calculations at each step of the loop do not depend at all on the results of the previous steps and **DISCL** is **0**. This is possible only when there is a mathematical formula for any member of the sequence, in which its value $\mathbf{A_n}$ depends only on the data known at the beginning of the calculation (**x**) and the step number **n** (iterative variable): $\mathbf{A_n = F(x, n)}$. Such formulas exist for arithmetic and geometric progressions, as well as for many other sequences. In this case the calculation of each member of the series is completely independent and is a local task, hence, they can be executed in any sequence and on any computing device, including in parallel. As for the Fibonacci sequence, to date this problem has not been solved, since the well-known Binet formula gives too large an error for it.

Two more problems remain to be solved:

– load balancing of computing devices;

– transfer of results to combine them into one amount.

The first problem is solved by distributing local problems between parallel computing devices according to the cyclic algorithm **round-robin** (Thakur V., Kumar S., 2014). Since the volume, and hence the time, of calculating the terms of the series is a monotonously increasing or decreasing function, this algorithm gives the best result.

The solution to the second problem is related to minimizing the time on switching between computing devices. To avoid having to synchronize the results multiple times during the calculation process, it would be optimal to form a local partial sum on each computing device from those members of the series that this device calculates. Then the switching will be performed once only at the very end of the calculations, when all partial sums will be fully formed.

A similar approach is applicable to some problems, such as numerical integration methods. If we do not go into details of the features of each method, it is highlighted their general part: the entire integration interval is divided into segments, on each of which numerical integration is implemented. The total value of the integral is then defined as the sum of the results at each segment. In doing so, to start calculations, it is enough to know the integrable function and the boundaries of a particular interval. That is, the calculation of each partial amount is completely independent of the others, hence, the integration of segments can be performed in any sequence, as well as in parallel on several computing devices.

Solving the load balancing problem in this case is much more complicated. All methods of Numerical Integration of Newton-Cotes quadrature rules are iterative, with the number of iterations depending on the required accuracy. The greater the number of segments into which the initial integration interval is divided, the more accurate the interpolation, the smaller the integration error. If Runge's rule (Runge Karl, 2019) is used to estimate the accuracy of calculations, the number of calculations performed on different segments will be different too. To equalize the volume of calculations, it is advisable to use the Gauss-Legendre method (Iserles Arieh, 1996) with unequal lengths of segments, however it is applicable to an odd number of nodes only and therefore to an odd number of parallel threads.

Minimization of switching time is also achieved by sending the result from each computing device to the master device one, where summation is performed in any sequence of the results obtained.

Experimental testing. The experimental test was performed on multi-core computers with 4-core processors Intel i3-10xxx, i3-7xxx, i5-8xxx, i5-6xxx and etc. To do this, the computers of students who participated in the testing his program were used. At the same time, to study multithreading as a consequence of multi-core, hyper-threading mode was disabled. The operability of the programs was checked by testing them on test cases, however, the main objective was to evaluate the temporal characteristics of the programs and their subsequent analysis.

MPI technology was used to implement multithreading because each kernel has its own cache memory L1 and L2, that is, with relatively small amounts of data refers to devices with distributed memory.

Internal parallelism is achieved by the fact that the thread rank is used in calculations as a parameter, due to which the universality of program execution by any core was achieved.

For example the sum of the members of a series is divided into two sums:

$$S_M(x) = \sum_{i=1}^{M} f(x,i) = \sum_{j=1}^{\frac{M}{N}} \sum_{k=0}^{N-1} f(x,k,j) \qquad (2)$$

The internal sum iterates through the threads of N computing devices, and the external sum – iterates, the number of which has decreased by N times.

As an initial task, calculations of the sums of Taylor decomposition series for various functions were simulated. Since the mathematical model of calculations was implemented in the form of iterative circles, the amount of calculations was determined by the required accuracy of the result. Thus, an array was obtained of temporary estimates of program execution for various mathematical functions, which were calculated with varying accuracy on different computers with multi-core processors, where the number of cores used in the calculations also changed.

Since all the processors that were used in the experiment had completely different performance characteristics, then there was no sense in comparing the resulting time estimates. The obtained estimates were processed by statistical methods, and their average values were used to calculate the indicator λ, which is proposed as an indicator of the effectiveness of multithreading.

Indicator λ characterizes the relative part of parallel calculations in their total number and used in Amdahl's formula (Amdahl Gene M., 1967). This indicator was chosen because it is more dependent on the on the internal parallelism of the program algorithm rather than the characteristics of the computing device on which the program is implemented.

To calculate the indicator λ, a transformation based on Amdahl's law was used:

$$= \frac{1-1/P_N}{1-1/N} \qquad (3)$$

where **P_N** – acceleration due to N parallel computing devices;
N – number of parallel computing devices.

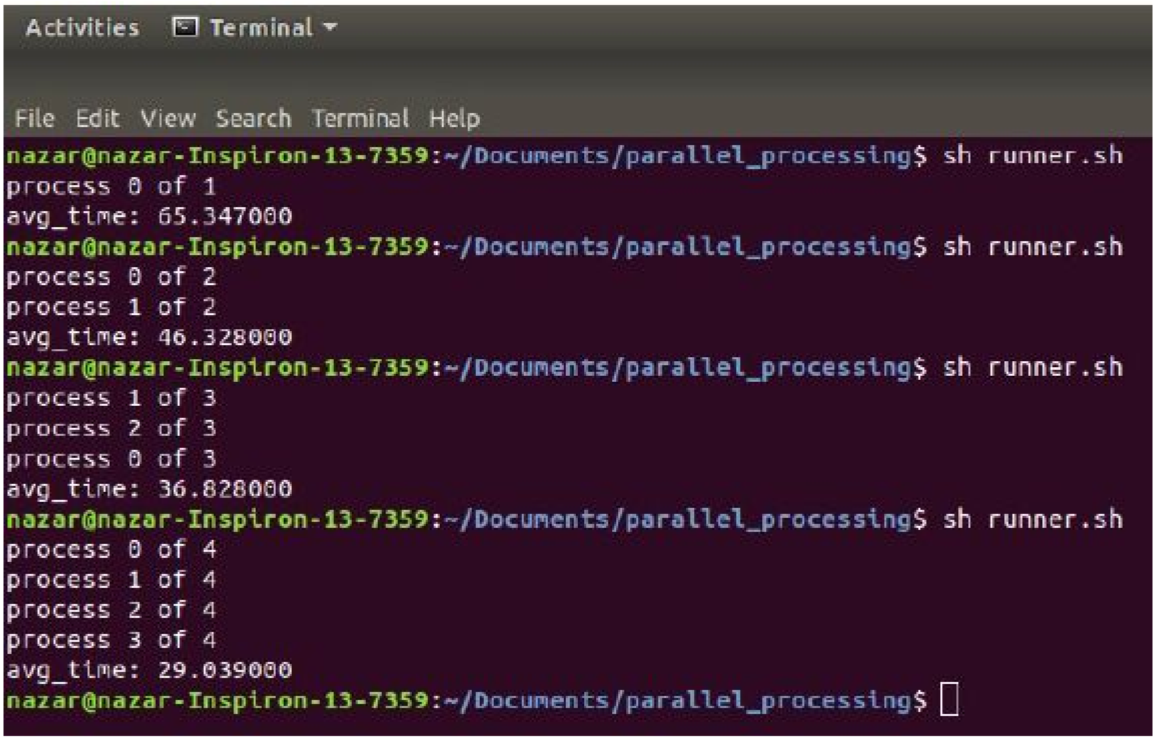
If for the same mathematical problem the calculations occur on the same computer, then we can get the function λ(N). If the value of λ(N) increases, it means that part of the parallel calculations increases, if it decreases, then the serial.

For example, for the results presented in Fig. 2, we get the following λvalues:

Table 1

Indicator λ for temporal estimates of the calculation of the hyperbolic cosine with an accuracy of 10^{-8}

Number of cores (N)	Time (ns)	λ
1	65.347	-
2	46.328	0.5829
3	36.828	0.6546
4	29.039	0.7408



```
nazar@nazar-Inspiron-13-7359:~/Documents/parallel_processing$ sh runner.sh
process 0 of 1
avg_time: 65.347000
nazar@nazar-Inspiron-13-7359:~/Documents/parallel_processing$ sh runner.sh
process 0 of 2
process 1 of 2
avg_time: 46.328000
nazar@nazar-Inspiron-13-7359:~/Documents/parallel_processing$ sh runner.sh
process 1 of 3
process 2 of 3
process 0 of 3
avg_time: 36.828000
nazar@nazar-Inspiron-13-7359:~/Documents/parallel_processing$ sh runner.sh
process 0 of 4
process 1 of 4
process 2 of 4
process 3 of 4
avg_time: 29.039000
nazar@nazar-Inspiron-13-7359:~/Documents/parallel_processing$ 
```

Fig. 2. Example of the formation of temporal estimates when calculating the value of the hyperbolic cosine with an accuracy of 10^{-8}

Table 2

Indicator λ for representation of various functions by Taylor series

Function	λ(N)			Function	λ(N)		
	N=2	N=3	N=4		N=2	N=3	N=4
Exponent	0,92	0,93	0,94	Geometrical Series	0,93	0,95	0,96
Natural Logarithm	0,78	0,83	0,85	Sine	0,87	0,88	0,90
Square Root	0,65	0,68	0,70	Cosine	0,88	0,89	0,89

Table 3

Indicator λ for numerical integration methods of Newton-Cotes

Function	λ(N)			Function	λ(N)		
	N=2	N=3	N=4		N=2	N=3	N=4
Of the Left Rectangles	0,78	0,84	0,86	Of the Middle Rectangles	0,84	0,85	0,85
Of the Right Rectangles	0,82	0,85	0,86	Of the Trapezes	0,86	0,89	0,89

Thus, we get a tool that allows us to compare the efficiency of multithreaded processing when solving, for example, various mathematical tasks on one computer or, conversely, the same task on computers with different characteristics.

To increase the repeatability of the result on the computers where the test was performed, almost all applications stopped, except for those that formed the software environment. However, there was some fluctuation in the indicators of the program execution time, so averaging was carried out, both in one experiment and the results of experiments. However it was not possible to achieve a complete elimination of background processes, which explains some growth in the λ score as the number of threads in the task increases within the physical cores of the processor, although, as a metric related only to the structure of the problem and its algorithm, it must be constant.

It can be stated that when calculating those functions where the volume of calculations is greater, the λ indicator increases significantly, this indicates that background processes outside the task are redistributed by the operating system manager and that they use shared resources less frequently. That is, in this way it is possible to indirectly evaluate active processes, including hidden ones, which are background for the task being solved, and the task itself can be a kind of indicator of the presence of hidden processes at the time of their activation.

Qualitatively similar results were obtained when testing the multithreading of the numerical integration problem. Here, the rank of the thread **k** defines for it the boundaries of the integration segment $\mathbf{a_k}$ and $\mathbf{b_k}$:

$$a_k = a + \frac{b-a}{N}k;\ b_k = \frac{b-a}{N}(k+1),\ \text{k=0, 1 … N-1} \quad (4)$$

As expected, there were difficulties in this task due to insufficient load balancing, because each thread was allocated an equal segment for integration. However, the convergence at each segment will be different, since it determines the type of integrable function, accordingly, the number of iterations will also vary. This causes the number of messages between threads to increase dramatically and cease to be predictable. This is because the task completion time is different for each thread, therefore synchronization is required through messaging to obtain an overall result.

Summary and Conclusion

1. Analysis of the evolution of the architecture of computer systems demonstrated, that the most promising is to increase the performance of calculations through the use of multithreading technologies.

2. Internal computation parallelism, as a prerequisite for implementing multithreading, can best be implemented for loops.

3. The «Depth of Inter-Step Communication in the Loop» (**DISCL**) indicator is proposed as a quantitative indicator of the possibility of optimizing cycles in the compiler.

4. The indicator λ is proposed as a criterion that characterizes the relative part of parallel calculations in their total volume and is invariant to the characteristics of the computer system. The possibility of applying this indicator has been tested experimentally.

BIBLIOGRAPHY:

1. Gordon Earle Moor. Cramming more components onto integrated circuits. *Electronics Magazine*. 1965. vol. 39(8). P. 114-117.
2. Flynn Michel. J. Very high speed computers. *IEEE*. 1966. 54(12). P. 1901 - 1909.
3. Wolf, Wayne Hendrix. Modern VLSI Design, 4th ed. Boston: Prentice Hall, 2002. 638 c.
4. Гергель В.П. Фурсов В.А. Лекции по параллельным вычислениям. Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2009. 164 с.
5. Гордеев Э.Н. Введение в теорию сложности алгоритмов. М: МГТУ им. Н.Э.Баумана, 2011. 49 с.
6. Chandra Rohit, Menon Ramesh, Dagum Leo, Kohr David, Maydan Dror, McDonald Jeff. Parallel Programming in OpenMP. Morgan Kaufmann, 2000. 229 c.
7. Gropp William, Lusk Ewing, Skjellum Anthony. Using MPI: Portable Programming with the Message Passing Interface. The MIT Press, 1999. 367 c.
8. Воеводин В.В., Воеводин Вл. В. Параллельные вычисления. Петербург: БХВ, 2002. 608 с.
9. Amdahl Gene M. The validity of the single processor approach to achieving large-scale computing capabilities. *In Proceedings of AFIPS Spring Joint Computer Conference, Atlantic City*, N.J., AFIPS Press. 1967. 30. P. 483-85.
10. Pavlov Valerii. Nullifying rules influence on speed in context free grammar LL(1). *Journal of Theoretical and Applied Computer Science. Polish Academy of Sciences, Gdansk Branch, Computer Science Commission*. 2016. 2. P. 3-15.
11. Aho Alfred V., Ullman Jeffrey D.. The theory of parsing, translation and compiling, Volume 2. Prentice-Hall, 1973. 542
12. Aho Alfred V., Sethi Ravi, Ullman Jeffrey D. Compilers: Principles, Techniques, and Tools. ADDISON-WESLEY, 1986. 796 c.
13. Thakur V., Kumar S., Load Balancing Algorithm An Analytical Study. *The IUP Journal of Computer Sciences*. 2014. VIII(2). P. 25-34.
14. Runge Karl. Analytische Geometrie der Ebene. Leipzig: B.G. Teubner, 1908. 217 c.
15. Iserles Arieh, A First Course in the Numerical Analysis of Differential Equations. Cambridge University Press, 1996. 480 c.

REFERENCES:

1. Moor G. E. (1965). *Cramming more components onto integrated circuits*. Electronics Magazine, vol. 39(8): 114-117. [in English].
2. Flynn M. J. (1966). *Very high speed computers*. IEEE, 54(12). 1901-1909. [in English].
3. W. Wolf. (2002). *Modern VLSI Design.* Fourth Edition. Prentice Hall.[in English].
4. Gergel V.P., Fursov V.A. (2009). *Lektcii po parallelnym vychisleniiam*. Samara: Izdatelstvo Samarskogo gosudarstvennogo aerokosmicheskogo universiteta. [in Russian].
5. Gordeev E.N. (2011). *Vvedenie v teoriiu slozhnosti algoritmov*. M: MGTU im. N.E.Baumana. [in Russian]
6. Chandra R., Menon R., Dagum L., Kohr D., Maydan D., McDonald J. (2000). *Parallel Programming in OpenMP*. Morgan Kaufmann. [in English].
7. Gropp W., Lusk E., Skjellum A. (1999) *Using MPI: Portable Programming with the Message Passing Interface*. The MIT Press. [in English].
8. Voevodin V.V., Voevodin Vl. V. (2002). *Parallelnye vychisleniia*. Peterburg: BHV [in Russian].
9. Amdahl Gene M. (1967). The validity of the single processor approach to achieving large-scale computing capabilities. *In Proceedings of AFIPS Spring Joint Computer Conference, AFIPS Press, 30*, 483-485. [in English].
10. Pavlov Valerii. (2016). Nullifying rules influence on speed in context free grammar LL(1). *Journal of Theoretical and Applied Computer Science. Polish Academy of Sciences, Gdansk Branch, Computer Science Commission*, 2, 3-15. [in English].
11. Aho A. V., Ullman J. D. (1973). *The theory of parsing, translation and compiling, Volume 2*. Prentice-Hall. [in English].
12. Aho A. V., Sethi R., Ullman J. D. (1986) *Compilers: Principles, Techniques, and Tools*. ADDISON-WESLEY. [in English].
13. Thakur V., Kumar S. (2014). Load Balancing Algorithm. An Analytical Study. *The IUP Journal of Computer Sciences, VIII(2)*, 25-34. [in English].
14. Runge K. (2019) *Analytische Geometrie der Ebene.* Wentworth Press. [in English].
15. Iserles Arieh (1996). *A First Course in the Numerical Analysis of Differential Equations*, Cambridge University Press [in English]