DOI 10.1007/s10559-023-00604-z
Cybernetics and Systems Analysis, Vol. 59, No. 5, September, 2023

CYBERNETICS

GLUSHKOV'S ALGORITHMIC ALGEBRAS AND AUTOMATED PARALLEL COMPUTING DESIGN

P. I. Andon,[1†] A. Yu. Doroshenko,[2] P. A. Ivanenko,[1‡] and O. A. Yatsenko[1††]

UDC 519.712, 004.4'24

Abstract. *An overview of the results obtained within the algebra of algorithms and tools for the automated development of programs for multiprocessor platforms is presented. Algorithmics is based on the theory of algorithmic algebras and is focused on solving a wide range of applied problems and developing software tools for automated design and synthesis of classes of algorithms and programs. The generality of algorithms is based on the variety of interpretations of algorithm schemes and provides the possibility of applying algorithms and their tools for solving problems related to various subject domains. The combination of algorithmics and rule rewriting technique made it possible to develop methods and tools aimed at automated design, transformation, synthesis, and tuning of programs for various platforms (multicore processors, graphics processing units, and field-programmable gate arrays).*

Keywords: *algebra of algorithms, parallel computating, program design and synthesis, software autotuning.*

INTRODUCTION

Currently, parallel computing on multiprocessor systems is the main source of ensuring high computing performance when solving complex scientific and technical problems. In recent years, the architecture of multiprocessor systems has become significantly more complicated due to the diversification of their components, which include multicore central microprocessors, graphic accelerators, multiprocessor clusters, and other structural elements. One of the promising areas of development and research of parallel and distributed computing systems is the construction of software abstractions in the form of formal languages and models using an algebraic-algorithmic approach in particular.

Algebra-algorithmic programming is a direction of the Ukrainian algebra-cybernetic school of automation of programming and design, which is based on the fundamental ideas and results of academician Viktor Mikhailovich Glushkov, which were obtained in the 60s of the last century from the theory of automata and formal transformations of programs to optimize calculations [1, 2]. Subsequently, this direction was developed by his colleagues, students, and followers, in particular, in the works of the team of the V. M. Glushkov Institute of Cybernetics of the National Academy of Sciences of Ukraine, headed by K. L. Yushchenko and G. O. Tseitlin [3–5] from the development of systems of algorithmic algebras (SAAs) and the "Multiprocessist" program synthesizer, as well as in the works of the team of the same institute headed by A. A. Letichevsky and Yu. V. Kapitonova [6–8] on the automation of the design of computing machines and algebraic programming. In recent decades, under the leadership of I. V. Sergienko, O. M. Khimich, and V. G. Tulchinsky,

[1]Institute of Software Systems, National Academy of Sciences of Ukraine, Kyiv, Ukraine, [†]*andon@isofts.kiev.ua*; [‡]*paiv@ukr.net*; [††]*oayat@ukr.net*. [2]Institute of Software Systems, National Academy of Sciences of Ukraine, Kyiv, Ukraine, and National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnical Institute," Kyiv, Ukraine, *anatoliy.doroshenko@gmail.com*. Translated from Kibernetyka ta Systemnyi Analiz, No. 5, September–October, 2023, pp. 3–15. Original article submitted March 17, 2023.

a powerful scientific and technical base was created in the institute for the research of information technologies and the application of high-performance computing, in particular, and their algebraic aspects [9–13]. Research on algebraic-algorithmic programming was also carried out at the Institute of Software Systems of the National Academy of Sciences of Ukraine by the team headed by P. I. Andon and A. Yu. Doroshenko in the direction of developing the theory, methods, and tools for automating the design of parallel programs [14, 15].

A specific feature of the Algorithmic Algebra (AA) [16] is the formalization of the processes of designing and synthesizing algorithms and programs. These objects are designed in terms of regular schemes, which are the algebraic representations in SAAs. The latter are used for the formalized design of algorithms, programs, and objects (abstract types of data and memory) in terms of uninterpreted and partially interpreted schemes, which are called processing strategies. Schematic presentation of objects forms knowledge about means of their design and generation. Along with the synthesis of knowledge in AA, there are means of forecasting and decomposition, as well as modeling and checking the efficiency of the behavior of models.

In [14, 15], the theory, methodology, and tools for the automated design, generation, and transformation of sequential and parallel programs based on the means of algorithmic algebras and rewriting rules are proposed. The purpose of this work is to review the results obtained over the last decade in the algebra of algorithms and corresponding programming tools for multiprocessor computing systems and networks.

1. ALGEBRAS OF ALGORITHMS AND TOOLS FOR DESIGNING HIGH-PERFORMANCE PROGRAMS

Similar to algebraic specifications in general, Glushkov's SAAs are oriented towards the analytical form of algorithm representation and the formalized transformation of these representations, in particular, to optimize algorithms according to the given criteria. Therefore, the SAA of Glushkov (Glushkov algebra) is a bibasic algebra $GA = \langle \{Pr, Op\};\ \Omega_{GA} \rangle$ where the bases Pr and Op are sets of logical conditions and operators defined on the information set IS, and Ω_{GA} is a signature of operations. Operators are mappings of the information (potentially partial) set into itself and logical conditions are predicates on the set IS, which acquire the values of three-valued logic $E_3 = \{0, 1, \mu\}$ where 0 is false, 1 is true, and μ is an uncertainty. The signature $\Omega_{GA} = \Omega_1 \cup \Omega_2$ consists of a system Ω_1 of logical operations that acquire values in the set Pr and a system of Ω_2 operations that acquire values in the set of operators Op.

Generalized Boolean operations belong to the logical operations of the system $\Omega_1 \subset \Omega_{GA}$, such as disjunction $u \vee u'$, conjunction, and $u \wedge u'$ objection $\overline{u}$, as well as the prediction operation $A \cdot u$ (of the left-hand side multiplication of the condition u on the operator A), which consists in checking the condition u after executing of the operator A.

The main operations, which are included in the system $\Omega_2 \subset \Omega_{GA}$ and generate operators from the basis Op, are the following:

— a composition $A * B$ is a binary operation on a set Op, which consists of the sequential application of operators $A \in Op$ and $B \in Op$;

— an alternative (branching) $([u]A, B)$ is a a ternary operation that depends on the condition $u \in Pr$ and the operators $A \in Op$ and $B \in Op$. This operation produces such an operator $C = ([u]A, B)$ that

$$C(m) = \begin{cases} A & \text{if } u(m) = 1, \\ B & \text{if } u(m) = 0, \\ W & \text{if } u(m) = \mu, \end{cases}$$

for each $m \in IS$ where W is an undefined operator;

— a loop $\{[u]A\}$ is a binary operation that depends on the condition $u \in Pr$ and the operator $A \in Op$. The essence of this operation is the cyclic application of the operator A for a false u as long as the condition u is not true.

A system of generators is fixed in the SAAs, which is a finite and functionally complete set of logical conditions and operators. With the help of this aggregate and the superpositions of operations included in Ω_{GA}, arbitrary operators and logical conditions are generated from the sets Pr and Op. Let us fix the SAA basis as $\mathfrak{A} \subset Pr \cup Op$.

A regular scheme is a superposition of operations included in the signature Ω_{GA} and of elements of the basis $\mathfrak{A}$.

Parallel calculations in SAAs are formalized using asynchronous disjunction operations, control points, and synchronizers [3, 14].

SAA tools form the basis of the SAA/I language [3–5, 14, 15] intended for multilevel structural design and documentation of sequential and parallel algorithms and programs. The advantages of the SAA/1 language are the ability to describe algorithms in a style close to natural language, as well as the availability of automated translation into an arbitrary programming language.

The main operator constructions of the language are the following:

— the composition of "*operator*1" and "*operator*2;"

— alternative IF '*condition*' THEN "*operator*1" ELSE "*operator*2" END IF;

— loop FOR (*counter* FROM start TO *fin*) "operator" END OF LOOP.

Representation of algorithms in the SAA/1 language is called a SAA scheme.

Note the conceptual closeness of AA to algebraic algorithms (AlgA) [18], and generative programming (GP) [19], as well as to formal methods of program development [20–22].

AlgA can be attributed to methods of top-down design of algorithms and programs, while GP can be attributed to to their top-down design. The specified AlgA is a formalized approach to the description of methods of processing mathematical (algebraic) objects. It combines various algebras and data processing algorithms used in the proof of basic theorems in the corresponding algebras.

The purpose of GP is to create different subject areas (SAs) and their integration using abstractions, biology, and programming ecology. An abstraction is an analytical representation of knowledge for a selected SA (for example, formulas in the corresponding algebras). The biology of programming in GP consists of the presence of "genetic" connections between close GPs. Ecology is a means of automating the construction of various SAs and their integration.

Similar to the above directions, AA is designed to solve the same problems, i.e., to formalize knowledge about SA by algebraic means [16, 17]. The biological aspect in AA is reflected in the theory of clones (families of algebras), and the ecological aspect is reflected in the system of tools for automating the design and synthesis of software. However, unlike AlgA and GP, AA considers three interdependent forms of presentation of algorithmic knowledge [14, 15], namely, the analytical in the form of an algebraic formula (regular scheme), the natural-linguistic, i.e., a text close to natural language, and the visual with the help of a graph (graph diagram).

Noting the closeness of AA to AlgA and GP, it is worth paying attention to the conceptual integrity of its paradigm, which largely determines the advantages of this paradigm compared to the above directions.

Instrumental means of AA support include dialog syntactically correct constructors (DSC-constructors), syntactic analyzers, object synthesizers (Fig. 1). The problems of analysis and synthesis were developed in the classical theory of automata. We note the need for decomposition (analysis) of objects for the possible subsequent collection (synthesis) of new knowledge (schematic patterns, strategies for processing, interpretation, etc.). Unlike traditional parsers, the DSC method is not focused on finding and correcting syntactic errors, but on preventing their appearance in the process of building an algorithm.

It is known that the synthesis process consists of the following two main stages:

— fixation of containers, i.e., basic concepts in the selected SA (semantic identifiers, interpretations, their implementations, etc.);

— generation of processing strategies and algorithms using programming languages and synthesis of programs based on the corresponding algorithmic algebras.

Schemes are forms of presentation of projected objects based on the application of certain metarules, such as convolution (abstraction), sweep (detailing), reinterpretation (convolution-unfolding), and transformation (application of equalities–identities, quasi-identities, and ratios characterizing the properties of operations that are included in the SAA signature).

The correctness of the modifications follows from the functional equivalence of the schemes obtained as a result of the modifications based on the use of the above metarules, which determines their semantic correctness.

Depending on the signature of the operations, the algebra underlying the selected design tools can be oriented to the sequential or parallel functioning of the designed objects. Multiprocessing is closely related to solving the problem of deadlocks (clinches) at the circuit level, and the occurrence of events is related to solving closed conditions for passing

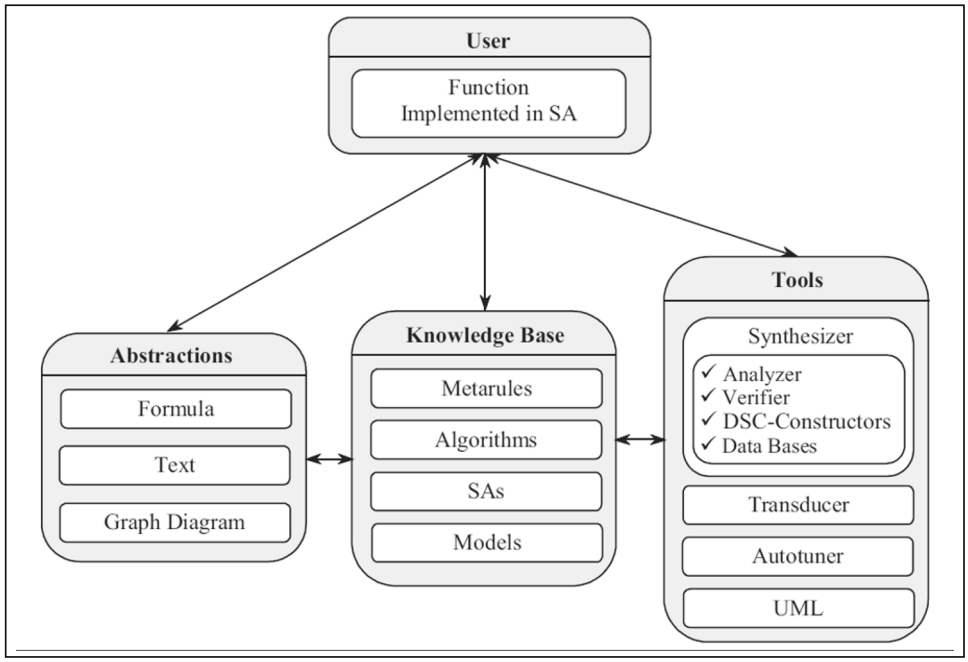

Fig. 1. Means of design and synthesis of programs used within AA.

the corresponding control points. Thus, the stated problem of clinches turns out to be related to the problem of identifying fictitious iterative constructions. Solving both problems requires recognizing the appropriate location of control points and synchronizers. In other words, it is necessary to ensure the verification of the execution of events and the means of delaying the calculation processes in interdependent parallel branches. The organization of parallelism at the level of schemes also depends on the use of means developed in the theory of closed and locally closed logical conditions (or monotonic operators and their generalizations).

Note that the instrumental means of designing objects that belong to various SAs have already been created and are being further developed. At the same time, the projected objects have a general structure given by transformations, as well as the above integrated forms of presentation, such as analytical, textual, and graphical. These forms reflect various complementary and instrumentally supported aspects of object design. In particular, the integrated toolkit for designing and synthesis of IDS (Integrated Toolkit for Designing and Synthesis of Programs) programs was developed [14]. The tools are intended for the design of sequential and parallel schemes of algorithms in SAAs and the generation of corresponding programs in the target programming languages like C++, Java, etc.

One of the important problems within the algebraic-algorithmic approach is increasing the adaptability of programs to the specific conditions of their use. In particular, it is solved by using a parametrically controlled generation of algorithms based on higher-level specifications, i.e., hyperschemes [14, 15], as well as means of automatic adjustment (autotuning) of programs to the target execution environment [23].

The object design process involves the integration of AA tools with other design tools (UML language, Rational Rose [24, 25], and object libraries). Note the complementarity of integrated tools from the point of view of their independent and joint use, in particular, for the development of special libraries aimed at supporting the synthesis of objects of various SAs.

In the mid-90s as part of the further development of AA, meta-algebras (clones) [14, 16] covering families of algorithmic algebras (structural, non-structural, etc.) were constructed and studied on the basis of known programming methods and technologies. Each of the algebras included in one or another clone can be associated with its own tools that correspond to the selected SAs and design methods. The first SAs created using the developing approach were associated with the algorithmization of a number of symbolic processing tasks (sorting, searching, and language processing). Then, the above integrated toolkit of design and synthesis of programs was developed, which provides automated construction of parallel multi-threaded and distributed programs in various SAs (in particular, meteorological forecasting).

Recent publications on the application of SAAs also include the generation of subject uniterms of formulas of the algebra of algorithms [26], the grammatical analysis of symbolic computations of expressions of the logic of statements [27], and the application of Johnson's algorithm for finding the shortest paths between all pairs of graph vertices [28].

Schemes of algorithms and interpretations form a knowledge base (KB), which reflects the essence of the selected SA. The KB also includes relations, identities, and quasi-identities, which are used in the process of transformation of schemes. In [29], the results of an experiment on the use of ontologies for presenting the base of algebraic-algorithmic knowledge are presented. With the help of ontology, the main objects of the developed program from the selected SA are described, namely, the data, functions, and relationships between functions.

The design and synthesis of objects with the help of algebraic schemes helps solving the following number of problems that also arise in GP:

— the formation of specialized libraries of basic concepts of the selected SA corresponds to the concept of generic programming, which provides the interpretation of the variables included in the designed schemes;

— object-oriented extensions are derived language constructions that ensure the fixation of object-oriented abstractions and can be interpreted as the creation of appropriate libraries-sections of the KB consisting of circuit projects (superpositions of signature operations of a given algebra of circuits) often used in this SA, i.e., analogs of extension libraries in GP;

— the specified extension libraries contain, in particular, optimization transformations, testing and debugging methods, editing, etc., effective for the selected SA;

— syntactic analysis of schemes is provided, firstly, by the presence of object analyzers in the corresponding design and synthesis language processors, and, secondly, by the presence of dialogue constructors of syntactically correct schemes or trees of their grammatical analysis. Note that ensuring syntactic correctness applies to all interdependent forms of object presentation (analytical, textual, and graphical-schematic);

— creation of synthesizers of objects and libraries of extensions (abstractions) oriented towards application in various SAs. Libraries of schemes (abstractions) significantly expand the capabilities of language processors and facilitate the understanding of program code. In addition, the algebraic (schematic) approach formalizes methods and technologies, is a tool for the integration of environments, and contributes to the creation and accumulation of various libraries that are part of the KB.

Thus, the algebraic approach to the design and synthesis of objects corresponds to the basic principles of GP and, at the same time, has a number of significant advantages, namely, the following:

— algebra is not only a convenient, understandable, accurate, and compact language for describing objects, but also oriented on formalized transformations of objects for their qualitative improvement according to selected criteria (memory, speed, hardware resources, etc.);

— algebraic formalism and tools based on it that are oriented towards the construction of objects (including not only schemas, but also input and target languages, as well as a variety of different SAs);

— the algebraic theory of clones was developed [14, 16], within which families of algebras of various types and subject orientations, which are the basis of super-high-level specification languages, were constructed in accordance with known object design methods. In the future, this means that application programmers have the opportunity to create their own convenient design languages and object synthesis tools that correspond to the selected SA and the target implementation environment.

2. APPLICATION OF ALGEBRA OF ALGORITHMS TOOLKITS

Instrumental means of automated design, generation, and optimization of algorithms and programs based on the use of SAAs have been applied to the development of software for general-purpose multicore processors [23], cloud platforms [29], graphics accelerators [30], programmable logic integrated circuits (PLICs) [31].

2.1. Designing and Automatic Tuning of Programs for Multi-Core Processors. In [23], a method is proposed that combines the means of algorithmic algebra and program autotuning, and is aimed at maximizing the efficiency of parallel programs in terms of execution time. The application of the approach is demonstrated on the example of designing and optimizing a parallel sorting program for execution on a multi-core processor.

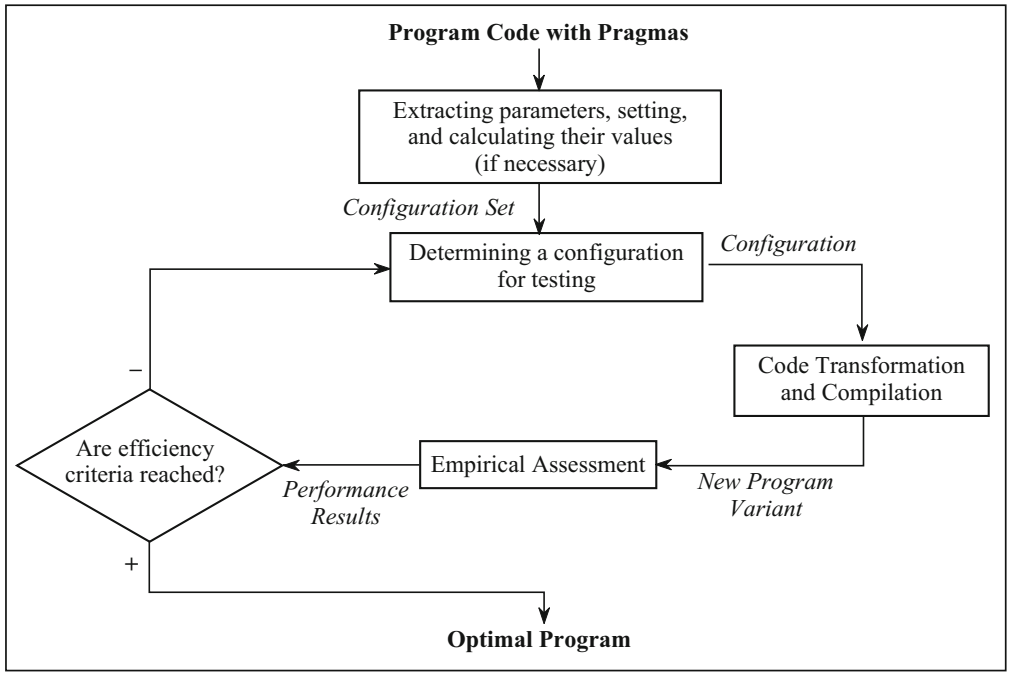

Fig. 2. Sequence of auto-tuning actions in the TuningGenie system.

Autotuning [32] is an automatic adjustment of the program to the execution environment, which uses an empirical approach to obtain a qualitative assessment of the optimized program (quality usually means speed and accuracy of the result). It automates the search for the optimal program variant from a set of predicted possibilities, executing each variant and measuring its performance on a given parallel architecture. The main advantage of this approach is a high level of abstraction, i.e., the program is optimized without knowing the details of the hardware implementation, such as the number of processor cores, cache size, and memory access speed. Instead, auto-tuning operates with high-level concepts from SAs, such as the number and dimension of independent problems or features of the problem-solving algorithm.

The developed auto-tuning system of TuningGenie program [23] accepts the source code of the program marked with pragmas (Fig. 2) at the input. Pragmas describe program configurations and transformations that affect its performance. Pragmas are manually defined by application developers using special-style Java language comments. First, information from all pragmas in the input program code is collected by the parser and based on it, all possible configurations of the application are generated. Next, for each configuration, a corresponding variation of the program is generated, and the speed is measured. Based on the obtained results, the optimal configuration is determined and the optimal version of the application is generated. For source code transformations, the TermWare system is used, which is based on the technique of rewriting rules [33], and processes program texts presented in the form of terms.

One of the pragmas is tuneAbleParam, which specifies the search area for the optimal value of a numeric variable. For example, for the threadCount variable and the selection limits [1…16] with a step of 1 are set as follows:

“Comment (tunableParam name=threadCount start=1stop=16 step=1)”
“Declare a variable (threadCount) of type (int) with initial value (1)”.

This pragma is primarily used for algorithms using geometric parallelisms as it allows us to easily choose the optimal decomposition of the calculation area, that is, the “granularity” of the algorithm. With the help of the tuneAbleParam pragma, we can also select some threshold value for changing the program calculation scheme.

The following is an example of a SAA schemes of a hybrid parallel sorting algorithm subject to further tuning with TuningGenie. The algorithm applies merging or insertion sort depending on the block size (insertionSortThreshold) of the input numeric array. The scheme contains two tuneAbleParam pragmas that specify the search area for optimal values of the insertionSortThreshold and mergeSortBucketSize variables. Based on the scheme, code is generated in Java, which is further optimized in TuningGenie.

```
"Parallel Hybrid Sorting (arr)"
==== "Comment(tuneAbleParam name=insertionSortThreshold start=10 stop=200
          step=10)";
     "Declare a variable (insertionSortThreshold) of type (int) with
     initial value (100)";
     "Comment(tuneAbleParam name=mergeSortBucketSize start=5000
          stop=1000000 step=5000)";
     "Declare a variable (mergeSortBucketSize) of type (int) with
     initial value (5000)";
     IF 'Length of the array (arr) is less or equal to (insertionSortThreshold)'
     THEN "insertionSort(arr)"
     ELSE IF 'Length of the array (arr) is less or equal to (mergeSortBucketSize)'
       THEN "sequentialMergeSort(arr)"
       ELSE "concurrentMergeSort(arr)"
       END IF
     END IF
```

Based on the above, increasing the adaptability of algorithms to the specific conditions of their use can also be carried out with the help of hyperschemes, which are parameterized algorithms for solving a certain class of problems. Setting parameter values and further interpretation of hyperschemes makes it possible to obtain algorithms adapted to specific application conditions [14, 15]

2.2. Programming Automation Tools for Graphic Accelerators. In [30], a software tool for optimizing calculations (LoopRipper) was developed, which enables parallelization of cyclic program operators in a semi-automatic mode for performing calculations on graphic accelerators. Data buffering, synchronized with the execution of the main loop, was carried out, and the tool, which was built using the TermWare rewriting rule system, is integrated with the IDS program design and synthesis toolkit.

The toolkit parallelizes the input sequential loop of the form

```
"SEQUENTIAL LOOP"
==== FOR (i_0 FROM 0 TO I_0)
       FOR (i_1 FROM 0 TO I_1)
       ...
       FOR (i_n FROM 0 TO I_N)
         "F(i, p^in(i), p^out(i))"
       END OF LOOP,
```

where $\underline{I_0, I_1, \ldots, I_N}$ are sets of index values $i_0, i_1, \ldots, i_N$; $N+1$ is a nesting depth of cycles; $\bar{i} = \{i_j \mid j = 0, \ldots, N\}$; $\overline{p^*(\bar{i})} = P^* = \{p_j^*(\bar{i}) \mid j = 0 \ldots \# P^*\}$, and $* \in \{in, out\}$ are sets of input and output variables describing the data processed in the loop; F is a function that processes data; loop iterations are independent.

Let T be the number of threads that is used to perform the core function of the graphics accelerator. The cycle obtained as a result of parallelization is as follows:

```
"PARALLEL LOOP"
==== FOR (e FROM 0 TO L)
       "fill(inBuf)";
       "push(inBuf)";
       "kernel(e, inBuf, outBuf)";
       "pull(outBuf)";
       "unpack(outBuf)"
       END OF LOOP,
```

where L is the number of kernel calls, which is chosen as the smallest possible and such that $L \cdot T \geq G$ (here, G is the total number of iterations in the initial cycle); *fill* is the function of filling the *inBuf* initial data buffer; *push* is the movement of the the initial data buffer to the device memory; *kernel* is a call to the core function (the latter

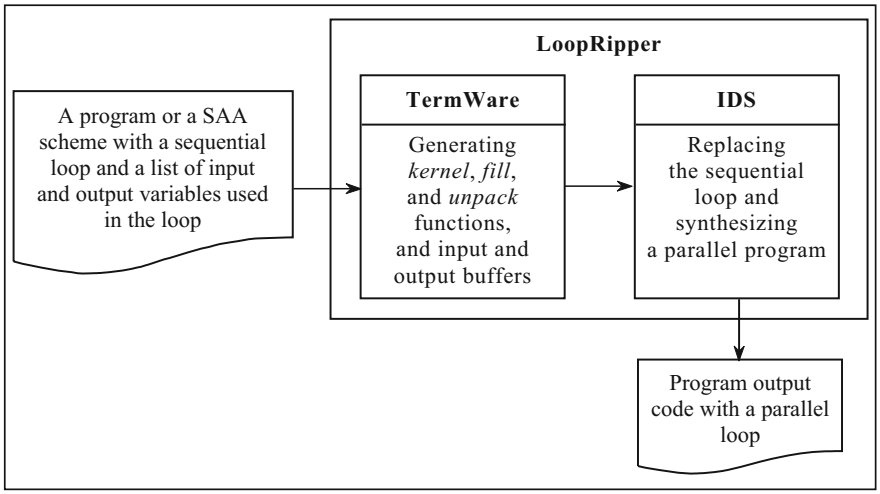

Fig. 3. Parallelization of the program using the LoopRipper toolkit.

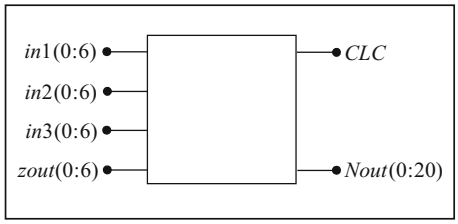

Fig. 4. An example of a genetic algorithm block.

contains a call to the body of the initial loop "$F(\bar{i}, \overline{inbuf_{id}(\bar{i})}, \overline{outbuf_{id}(\bar{i})})$" where *id* is a stream number), *pull* is a function that moves the processed data from the device memory to the processed data buffer *outBuf* , and *unpack* is copying data from the processed data buffer into the appropriate variables.

Figure 3 shows the process of program parallelization using the LoopRipper toolkit.

Using the lists of input and output variables, LoopRipper generates kernel, fill, and unpack functions, from which a parallel program is composed using the program design and synthesis toolkit. Functions are generated by replacing parameters with input and output data buffers and recalculating iterators in a given initial loop. Thus, the user is required to provide a loop and parameter lists.

The proposed method is applied to the parallelization of a sequential cycle with a nest depth equal to four, and it is included in the program of numerical weather forecasting, namely, the cycle of interpolation of initial data in local grid nodes. The developed system was tested on a heterogeneous multicore cluster.

2.3. Parallel Programming Automation Tools for PLIC. In [31], means of automated design and generation of programs for PLIC based on the algebra of algorithms are proposed, and their application to the development of a genetic algorithm and an artificial neuron is demonstrated. The technology of developing applications for PLIC is based on the presentation of the algorithm in the hardware description language, such as VHDL [34], and the automatic translation of this description into a specification at the level of logic tables and other functional components of the PLIC. The VHDL language provides a high-level abstraction for hardware description by having a set of predefined data types and the ability to create user-defined and hierarchically organized data types based on the basic ones built into the language. Elementary functions in the PLIC are usually implemented as separate projects or modules containing information about the data transfer rate and the internal structure of the system.

Parallel computations characteristic of neural networks are efficiently implemented using the VHDL language. Each process in the VHDL has a list of signals whose change causes the process to run. Based on this, neurons (individual processes) are naturally combined into a network where the outputs of the previous layer of the network trigger the processes corresponding to the next layer. Each neuron is represented as a separate block consisting of several parallel processes, and a neural network is a multiprocessor system.

Let us consider the genetic algorithm block (Fig. 4) as an example. It consists of the following ports: *in*1, *in*2, and *in*3, which receive such input signals as *zout*, which stores the value obtained as a result of training and *Nout* (output port) showing the current value of the output after each training iteration. The *CLC* signal has a delay of one picosecond. The description of the ports of this block is represented by the following SAA scheme:

ENTITY *genVHDL* is
"Signal (*in*1) direction (*in*) of type (*integer*) and range (−100 to 100) with initial value (63)";
"Signal (*in*2) direction (*in*) of type (*integer*) and range (−100 to 100) with initial value (− 82)";
"Signal (*in*3) direction (*in*) of type (*integer*) and range (−100 to 100) with initial value (70)";
"Signal (*zout*) direction (*in*) of type (*integer*) and range (0 to 100) with initial value (77)";
"Signal (*CLC*) direction (*inout*) of type (*bit*) with initial value ('1')";
"Signal (*Nout*) direction (*inout*) of type (*integer*) and range (−1048575 to 1048576);
END OF ENTITY

The block implements the operators of crossover, the mutation, and selection of the next generation of chromosomes. The process implemented by the neuron is designed in the form of the following SAA scheme:

PROCESS *neuron* (*start*) is
DECLARATIONS (
"Variable (*LocalOut*) of type (*integer*) and range (0 to 1023)";
"Variable (*lin*) of type (*integer*)");
$(LocalOut := 100)$;
$(lin := (in1 * W(1) + in2 * W(2) + in3 * W(3))/16)$;
FOR (*a* FROM 0 TO 49)
IF $(\text{abs}(lin) >= x(a))$ AND $(\text{abs}(lin) < x(a+1))$
THEN $(LocalOut := 50 + a)$; EXIT LOOP;
END IF
END OF LOOP;
IF $(lin < 0)$ THEN $(LocalOut := 100 - LocalOut)$;
END IF;
IF (NOT (*finishTeaching*)) THEN
$(lout(i) := LocalOut)$;
$NFinish <= \text{NOT } NFinish$;
END IF;
$Nout <= LocalOut$;
END OF PROCESS.

Here, *LocalOut* is the current output of the neuron, which is further compared with the expected result of the *zout* variable, $W(1)$, $W(2)$, and $W(3)$ are synaptic weights of the neuron, and *i* is the number of the chromosome, on the basis of which the weighting coefficients were formed before starting the process that forms the output of the neural network.

Code generation in VHDL language was performed on the basis of the constructed SAA schemes in the integrated toolkit.

CONCLUSIONS

The article provides an overview of the results obtained within the scientific direction of algebra of algorithms and means of automating the design of high-performance programs for multiprocessor platforms initiated by academician V. M. Glushkov. Algorithmics is based on the theory of algebra of algorithms and is focused on solving a wide range of applied problems and developing software tools for automated design and synthesis of classes of algorithms and programs. The generality of algorithms is based on the variety of interpretations of algorithm schemes and provides opportunities for the application of algorithms and its tools for solving problems in various subject areas. The efficiency of such an application largely depends on the depth of understanding of the semantics of the developed algorithms. Only under this condition is it possible to adequately interpret algorithmic tools and their efficiency during the design of

algorithms and programs. The combination of algorithms and the technique of rewriting rules made it possible to develop methods and tools aimed at automated design, transformation, synthesis, and customization of programs for various platforms (multicore processors, graphics accelerators, and programmable logic integrated circuits).

REFERENCES

1. V. M. Glushkov, Synthesis of Digital Automata [in Russian], Fizmatgiz, Moscow (1962).
2. V. M. Glushkov, "Automata theory and formal microprogram transformations," Cybern. Syst. Analysis, Vol. 1, No. 5, 1–8 (1965). https://doi.org/10.1007/BF01071417.
3. V. M. Glushkov, G. E. Tseitlin, and E. L. Yushchenko, Symbolic Multiprocessing Methods [in Russian], Naukova Dumka, Kyiv (1980).
4. V. M. Glushkov, G. E. Tseitlin, and E. L. Yushchenko, Algebra, Languages, and Programming, 3rd ed. [in Russian], Naukova Dumka, Kyiv (1989).
5. E. L. Yushchenko, G. E. Tseitlin, V. P. Gritsai, and T. K. Terzyan, Multilevel Structured Program Design: Theoretical Foundations and Tools [in Russian], Finansy i Statistika, Moscow (1989).
6. V. M. Glushkov, Yu. V. Kapitonova, and A. A. Letichevsky, Computer-Aided Design of Computers [in Russian], Naukova Dumka, Kyiv (1975).
7. Yu. V. Kapitonova and A. A. Letichevsky, Mathematical Theory of Computer System Design [in Russian], Nauka, Moscow (1988).
8. A. A. Letichevsky, J. V. Kapitonova, and S. V. Konozenko, "Computations in APS," Theor. Computer Sci., Vol. 119, 145–171 (1993).
9. I. V. Sergienko, S. L. Kryvyi, and O. I. Provotar, Algebraic Aspects of Information Technologies [in Ukrainian], Naukova Dumka, Kyiv (2011).
10. I. V. Sergienko, Topical Directions of Informatics, In Memory of V. M. Glushkov, Springer Optimization and Its Applications, Vol. 78, Springer, New York (2014). https://doi.org/10.1007/978-1-4939-0476-1.
11. M. R. Petryk, O. M. Khimich, and I. V. Boyko, High-Performance Methods of Modeling and Identification of Complex Processes and Objects in Multicomponent Heterogeneous Environments [in Ukrainian], V. M. Glushkov Institute of Cybernetics, NAS of Ukraine, Kyiv (2020).
12. O. M. Khimich, V. I. Mova, O. O. Nikolaichuk, O. V. Popov, T. V. Chistjakova, and V. G. Tulchinsky, "Intelligent parallel computer with Intel Xeon Phi processors of new generation," Nauka Innov., Vol. 14, No. 6, 66–79 (2018). https://doi.org/10.15407/scin14.06.066.
13. A. L. Golovynskyi, A. L. Malenko, I. V. Sergienko, and V. G. Tulchinsky, "Power efficient supercomputer SCIT-4," Visn. Nac. Akad. Nauk Ukr., No. 2, 50–59 (2013).
14. F. I. Andon, A. E. Doroshenko, G. E. Tseitlin, and E. A. Yatsenko, Algebra-Algorithmic Models and Parallel Programming Methods [in Russian], Akademperiodyka, Kyiv (2007).
15. P. I. Andon, A. Yu. Doroshenko, K. A. Zhereb, and O. A. Yatsenko, Algebra-Algorithmic Models and Methods of Parallel Programming, Akademperiodyka, Kyiv (2018). https://doi.org/10.15407/akademperiodyka.367.192.
16. G. E. Tseitlin, Introduction to Algorithmics [in Russian], Sfera, Kyiv (1998).
17. L. Zakhariya, "Algebra-algorithmic approaches in the subject areas and synthesis description of software environments for them," Bulletin of Lviv Polytechnic National University, Ser. Information Systems and Networks, Iss. 832, 376–384 (2015).
18. P. Naudin and C. Quitté, Algorithmique Algébrique: Avec Exercices Corrigés, Masson, Paris (1992).
19. K. Czarnecki and U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, Boston (2000).
20. M. Roggenbach, A. Cerone, B.-H. Schlingloff, G. Schneider, and S. A. Shaikh, Formal Methods for Software Engineering: Languages, Methods, Application Domains (Texts in Theoretical Computer Science, An EATCS Series), Springer, Cham (2022). https://doi.org/10.1007/978-3-030-38800-3.

21. J. Wang and W. Tepfenhart, Formal Methods in Computer Science, Chapman and Hall/CRC, New York (2019). https://doi.org/10.1201/9780429184185.
22. D. Sannella and A. Tarlecki, Foundations of Algebraic Specification and Formal Software Development, Springer Berlin–Heidelberg (2012). https://doi.org/10.1007/978-3-642-17336-3.
23. A. Doroshenko, P. Ivanenko, O. Novak, and O. Yatsenko, "A mixed method of parallel software auto-tuning using statistical modeling and machine learning," in: V. Ermolayev, M. Suárez-Figueroa, V. Yakovyna, H. Mayr, M. Nikitchenko, and A. Spivakovsky (eds.), Information and Communication Technologies in Education, Research, and Industrial Applications, ICTERI 2018, Communications in Computer and Information Science, Vol. 1007, Springer, Cham (2019), pp. 102–123. https://doi.org/10.1007/978-3-030-13929-2_6/.
24. S. Sundaramoorthy, UML Diagramming: A Case Study Approach, Auerbach Publications, Boca Raton (2022). https://doi.org/10.1201/9781003287124.
25. W. Boggs and M. Boggs, Mastering UML with Rational Rose, Sybex, Alameda, CA (2002).
26. A. Vasyluk and T. Basyuk, " Synthesis system of algebra algorithms formulas," Bulletin of Lviv Polytechnic National University, Ser. Information Systems and Networks, Iss. 9, 11–22 (2021). https://doi.org/10.23939/sisn2021.09.011.
27. V. V. Lytvyn, I. O. Bobyk, and V. A. Vysotska, "Application of algorithmic algebra system for grammatical analysis of symbolic computation expressions of propositional logic," Radio Electronics, Computer Science, Control, No. 4, 77–89 (2016). https://doi.org/10.15588/1607-3274-2016-4-10.
28. S. D. Pogorilyy and M. S. Slynko, "Research and development of Johnson's algorithm parallel schemes in GPGPU technology," Problems in Programming, No. 2–3, 105–112 (2016).
29. A. Yu. Doroshenko, O. A. Yatsenko, and O. M. Ovdii, "Ontological and algebra-algorithmic tools for automated design of parallel programs for cloud platforms," Cybern. Syst. Analysis, Vol. 53, No. 2, 323–332 (2017). https://doi.org/10.1007/s10559-017-9932-8.
30. A. Yu. Doroshenko, O. A. Yatsenko, and O. G. Beketov, "Algorithm for automatic loop parallelization for graphics processing units," Problems in Programming, No. 4, 28–36 (2017).
31. A. Doroshenko, V. Shymkovych, O. Yatsenko, and T. Mamedov, "Automated software design for FPGAs on an example of developing a genetic algorithm," in: Proc. 17th Intern. Conf. "ICT in Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer," ICTERI 2021 (Kherson, Ukraine, 28 Sept – 2 Oct, 2021), Vol. 1: Main Conference, PhD Symposium, Posters and Demonstrations, CEUR-WS (2021), pp. 74–85.
32. J. Durillo and T. Fahringer, "From single- to multi-objective auto-tuning of programs: Advantages and implications," Sci. Program., Vol. 22, No. 4, 285–297 (2014). https://doi.org/10.3233/SPR-140394.
33. R. Shevchenko, "Context term calculus for rewriting systems," Problems in Programming, No. 2–3, 21–30 (2018).
34. A. P. Godse and Dr. D. A. Godse, VHDL Programming: Concepts, Modeling Styles and Programming, Amazon Digital Services LLC, Seattle (2020).