Bull. Nov. Comp. Center, Comp. Science, 45 (2021), 29-48
@ 2021 NCC Publisher

Functional programming for parallel computing

L.V. Gorodnyaya

Abstract .The paper is devoted to modern trends in the application of functional programming to the problems of organizing parallel computations. Functional programming is considered as a meta-paradigm for solving the problems of developing multi-threaded programs for multiprocessor complexes and distributed systems, as well as for solving the problems associated with rapid IT development. The semantic and pragmatic principles of functional programming and consequences of these principles are described. The paradigm analysis of programming languages and systems is used, which allows assessing their similarities and differences. Taking into account these features is necessary when predicting the course of application processes, as well as when planning the study and organization of program development. There are reasons to believe that functional programming is capable of improving program performance through its adaptability to modeling and prototyping. A variety of features and characteristics inherent in the development and debugging of long-lived parallel computing programs is shown. The author emphasizes the prospects of functional programming as a universal technique for solving complex problems burdened with difficult to verify and poorly compatible requirements. A brief outline of the requirements for a multiparadigm parallel programming language is given.

Keywords: Functional programming, parallel computing, programming languages, programming system, programming paradigm, multi-paradigm

Introduction

A large number of parallel programming languages and systems have already been created to solve the numerous problems of developing multithreaded programs for multiprocessor complexes and distributed systems. Interestingly, the ideas of functional programming and their practical application to the analysis of the problems and methods of organizing parallel computations are becoming more popular. Recent studies in functional programming have focused on finding more efficient solutions to parallel programming problems [1].

The results of the analysis of the modern trends in functional programming allow us to consider it as a methodology for solving the problems of organizing parallel computations. The paradigm analysis of languages and programming systems is involved. This allows us to describe the semantic and pragmatic principles of functional programming and their consequences. Taking into account the paradigmatic features is useful when predicting program application processes and when planning their study and development. Functional programming helps us to improve the performance of programs by preprocessing their prototypes Functional programming is especially promising as a universal method for solving complex problems burdened with difficult-to-verify and poorly compatible requirements. It makes it possible to consider many features and characteristics inherent in the development and debugging of parallel computing programs.

The presentation begins with a brief description of a number of parallel computing paradigms supported in well-known programming languages. Then, the author dwells on the basic principles of functional programming and their concretization, which arise when functional programming methods are transferred to parallel computations. Also, we have considered several parallel computing paradigms supported in a notable number of programming languages. In conclusion, the requirements for a modern parallel programming language are described.

1. Paradigmatic characteristic

It should be noted that many parallel programming languages have already been created. A possible challenge for the developers of parallel computing programs is the hidden multiplicity of paradigms [2-4]. The developers have to consider simultaneously a variety of aspects at different angles in different paradigms. Usually, a separate paradigm is convenient for solving isolated subtasks if a full set of subtasks cannot be solved in the common environment. Obviously, the tasks of scaling computations, synchronizing interactions in multi-threaded programs, representing natural asynchronous parallelism and achieving high program performance are different. Moreover, it is a separate task to learn the phenomena of parallelism in order to give specialists an intuitive idea of the methods for solving the problems depending on unusual phenomena. Recent research in functional programming is aimed, in fact, at finding more stable solutions to parallel programming problems, allowing their correctness to be checked on models and debugging on prototypes.

1.1. Paradigms of parallel programming languages

To solve the main of these tasks in different paradigms, many languages and programming systems supporting parallel computing have been created (see Table 1).

Table 1.
Support for parallel computing problems and paradigms in different programming languages

problems	paradigms	programming languages and tools
Scaling	**Multiprocessor** programming	VHDL, XC, SIGMA, bash, Occam, mpC, EL'-76, Limbo, Kotlin, MPI
Thread Synchronization	**Sync** programming	APL, VAL, Sisal, Alef, E, X10, LuNA, Charm, Go, Java, Scala, Rust, Pifagor, OpenMP
Problems Statement	**Asynchronous** programming	BARS, Haskell, Erlang, JavaScript, Python, C#
Programs Performance	**High performanc**e programming	Setl, HPF, G, Sparkel, mpC, Sanscript, D, Rest, F#
Concurrency Familiarization	**Instructional** programming	Logo, Robik, Karel, OZ, A++, Sinkhro, Lsl

Thus, for each of the hard-to-solve problems of parallel computing, a separate paradigm convenient for its solution has been formed and a number of programming languages to support it have been created. The difference between paradigms is in prioritizing the means and methods used when solving a separate problem; different problems require a different ordering. Typically, only one paradigm is used at each stage of program development. Accordingly, there is only one main paradigm in every programming language. The requirements for solving rather complex problems of parallel computing are associated with certain difficulties. This implies the need to use different paradigms at certain stages of their creation and phases of their lifecycle, especially for long-lived programs. During the transition to programming technology, it is important to obtain practical results, which requires the support of a full range of paradigms used at the different stages of program development its lifecycle is formed. As a rule, to facilitate the simultaneous use of different paradigms in solving the same problem, multilingual systems are created that enable us to move from one paradigm to another without the need to learn different interfaces and systems. It is hence expedient to create a multi-paradigm parallel computing language which will simultaneously support all the main paradigms of parallelism. Importantly, at each stage of program development only one paradigm should be used, with its relatively small set of tools and methods within one way of thinking. The use of the BARS and Haskell languages shows that in these cases it is convenient to decompose the programming language definition into separate sublanguages supporting the main paradigms or monads aimed at specific models for developing and presenting programs. Then each paradigm has its own content of the categories of semantic systems and its own order of their role in the programming process [5, 6].

Multiprocessor programming tools usually rely on data structures and characteristics of the available architecture, including the underlying multiprocessor complex and relationships between its elements. Operating the complex allows one to initiate the functioning of separate processors, as well as their blocking, resumption and cancellation. During these processes, data exchange is possible via certain protocols, the final results of which can be considered as the goal of the program. Decision making begins with defining the space of possible multiprocessor complexes, which can be considered as a special kind of memory with its own discipline of functioning and interaction of elements. The next step is the selection of suitable configurations and appropriate structuring of the iteration space of the processes intended to be executed on separate processors. Then the resulting process control scheme is filled with actual actions performing calculations. Generally, routines without recursion are preferable since they are free from the complexities of computation control and hard-to-predict side effects of shared memory. A multiprocessor program being built allows one to reconfigure a multiprocessor complex [1] dynamically.

In synchronous multi-threaded programming, clear computation control schemes are determined and regular sections and typical program models that are convenient for parallelization are identified. Usually, memory fragments free from side effects are found and non-imperative control of the execution time of program threads is allowed, which takes into account the hierarchy of the program control scheme and some timing relations. Decision-making begins with the choice of standard control schemes using the concept of "iteration space," which can be structured depending on the data distribution and methods of storage, which can affect the efficiency and discipline of multi-level memory processing. Iteration control can use arbitrary predicates. The control scheme over the stream iteration space is filled with the fragments relatively easy to debug, possibly debugged in advance or programmed independently. To return the results of the program, special means are allocated for generating the results of calculations obtained on single-level threads of a multi-threaded program. Thus, a multi-threaded program is obtained with a dynamically variable space of threads over local memory with the possibility of episodic synchronization of their individual fragments[2].

Asynchronous programming is aimed at representing independent program elements reflecting the nature of the problem being solved, which can be the basis of maximum parallelization, provided that special schemes for computing organization are identified, taking into account the characteristics of the available equipment. Decision-making begins with the choice of action control schemes and with the definition of conditions to start them. Actions can use one or another discipline of processing different types of memory. An implicit and programmable variety of memory access disciplines are supported,

1 https://mooc.tsu.ru/mooc-openedu/mpi/ "Parallel programming using OpenMP and MPI"

2 https://habr.com/ru/post/121925/ "MPI Basics"

including a heterogeneous memory hierarchy and computation schemes designed to include the fragments that are procedures or library units. The result is a program which is a synchronizing network diagram which asynchronously controls the execution of actions provided they are ready for execution. High-performance programming requires a transition from a single program run to the consideration of the prospects for its repeated use and optimization. This makes it possible to take advantage of the underused capacities of a multiprocessor complex and to execute separate program fragments for the upcoming calculations taking into account the data which may be required in the forthcoming runs of the program. Decision-making begins with computation schemes and models, possibly over shared memory, but with local memory priority.

For the problems of program performance improvement, debugging control should take into account the need to re-execute a program during testing and take advantage of the inheritance of results between program runs and of the comparison of the measured characteristics of the program versions performance. Finally, we have several improved versions of the program, and the one chosen may be better at taking into account the conditions of application, including the configuration of a multiprocessor complex and the requirements for program quality.

Educational programming is intended to fill the gaps in recommendations for imperative solutions to any problems. Such gap points make it difficult to study programming methods in general, even more so in parallel computing. The development of game programs (game writing), such as robot visualization, can encourage learning all paradigms of parallel computing. This is sufficient reason for multi-paradigm natureof educational programming languagers, which usually support the means of parallel computations representation [7-9].

Long-lived programming languages, as well as new programming languages, are usually multi-paradigm. Successful parallel programming practice requires support for a full range of parallel computing paradigms. Their development, replenishment and application should be considered as necessary, ensuring the transition to the next paradigm without changing the programming languages and system environment.

2. Principles of functional programming

While the number of programming languages is now growing rapidly, the paradigms are not many. The basic ideas of functional programming were proposed by John McCarthy [3] in his early works on artificial intelligence [10]. Functional programming is one of the first paradigms aimed not only at the efficient implementation of the algorithms developed, but also at solution of new information processing problems with a research component [10-19]. This situation allows for a fairly complete comparison and selection of characteristics for a clear differentiation of paradigms and understanding the

3 In December 1968, John McCarthy read a series of lectures on the Lisp language at the Computing Center of the Siberian Branch of the Academy of Sciences in the office of A.P. Ershov.

reasons for their diversity. When considering the results of the task analysis, means and methods of parallel computations organization, one can draw attention to the obvious variety of problem settings and corresponding priorities in decision-making at the different stages of program development and debugging. A methodology of programming language comparison presented in [20-24] is based on an informal definition of the term "programming paradigm" given in [25], stating that paradigm comparison requires highlighting the distinctive testable features. When making decisions at the different stages of program development and debugging, we have assumed ordering priorities to be these features. This should be taken into account when predicting the complexity of the processes of program application, starting from planning, investigation and organization of the development of long-lived programs [26-29].

A general description of the functional programming paradigm often begins with the assertion that its characteristic feature is that the same formulas in the same context have the same meaning. This leads to the elimination of assignments, global variables, side effects, and control transfers.

First of all, it should be noted that the concept of "context" is somewhat ambiguous. It is both a program fragment, sometimes called the scope of existence, or visibility, and a context table of the correspondence between the symbols used in this fragment and their meanings. A programming language may have two or more context tables for one fragment - static and dynamic - and, in addition, global and local contexts. Programming systems for one language can solve the problem of the order of iterations in different ways through the context tables to determine the value of the formula. Moreover, this order can be changed by the options from the program or task for its compilation. If a line segment between two adjacent assignments is considered as context, then this characteristic does not distinguish the functional programming paradigm from other paradigms. Meanwhile, in the process of compilation, precisely this kind of program decomposition is practiced to solve memory allocation problems. When compared to functional programming, each such interval can be thought of as a separate context where each value is associated with its own local variable. In general, the above formula will be valid in any paradigm allowing programs to be represented as single assignment sections.

As for global variables, side effects and control transfers, McCarthy also noted that the outermost local variable plays the role of a global one [10]. The purely functional programming language Haskell introduced the concept of monads to exploit side effects. In many functional programming languages, the lack of control transfers is compensated by exception, extension and continuation mechanisms.

Functional programming usually implies the support of a number of semantic and pragmatic principles that contribute to the creation of functional models at the stage of computer experiments useful in solving new problems. When developing a program, the programmer follows semantic principles. Pragmatic principles are provided by the programming system, freeing the programmer from insignificant solutions independent of the nature of the problem. Most of these principles were laid down by J. McCarthy in the first implementations of the Lisp language [10].

2.1. Semantic principles

Functional programming supports semantic principles for algorithm representation, such as universality, self-applicability, and parameter independence.

Universality. The concepts of "function" and "value" are represented by the same symbols as any data for computer processing. Each function applied to any given data produces a result or a diagnostic message in finite time. A historically related concept is the stored program principle.

This principle allows one to build representations of functions from their parts and calculate the parts as data arrive and are processed. In principle, there are no restrictions on the manipulations with language means, functions from the definition of the language semantics, constructions for the implementation of the language in a programming system, or program expressions.

Everything needed for the implementation of a programming language can be helpful when the language is used. This determines the openness of functional programming systems. Strictly speaking, programming, unlike mathematics, does not deal with any values or functions but with data that can represent values or functions. This was noted long ago by S.S. Lavrov [14, 20].

There are no obstacles to processing function representations in the same way as data are processed. Therefore, representations of functions can be built from their parts – symbols. They can even be formed during the computation process and when processing the information about them. Any information necessary for computer processing can be represented using symbols [10].

When a program is compiled, memory is allocated for functions, variables, and constants. The efficiency of such a distribution depends on the due consideration of the specifics of the basic means for machine code processing, usually formulated as a data type convenient for computer processing but somewhat contradictory to the principle of universality.

An idea similar to that of a stored program was first formulated in the description of Charles Babbage's analytical engine. A hundred years later, it was implemented in the computers of Konrad Zuse and in the definition of the Turing machine, and later proclaimed in John von Neumann’s architecture [30]. As a result, a code or symbolic representation of information is possible, in which there is no fundamental difference in the nature of data for depicting values and functions. Consequently, there are no obstacles to processing function representations by the same means as data are processed.

Self-applicability. Function representations can use themselves directly or indirectly, which allows for a construction of clear concise recursive symbolic forms. Both values and functions can have a recursive representation.

Examples of self-applicability are given by many mathematical functions, especially recursive ones, such as factorial, Fibonacci numbers, series summation and many other functions defined by mathematical induction. In programming technology, the methods of step-by-step, or continuous, development of programs, as well as extreme programming, have some similarities. These methods reduce the organization of the programming process to a series of steps, each of which provides either a workable part of a program or a tool for performing the next development steps. Thanks to this principle, it is possible to apply in practice the method of step-by-step program development, in which a minimal kernel is selected and implemented, and then the steps of its expansion are performed. For the first implementations of the Lisp language, the interpreter and compiler were described in Lisp itself, and the descriptions took less than two pages [10]. The same is true for the development of the C language [31].

Equal rights of parameters. The order and method of evaluating the parameters is irrelevant. The function parameters are independent of each other.

One can note that when a function is called, its parameters are calculated at the same level of hierarchy and in the same context. Some of the parameters are calculated before a function call, while others can be calculated later but in the same context. Therefore, a representation of any highlighted formula from a function definition can be turned into a parameter of this function. This means that parts of the function representation can be calculated depending on the intermediate results, and functions can be constructed taking into account the conditions of their use, in particular, the location of their definitions and calls at different levels of the program representation hierarchy. Any symbolic form in a function definition can be extracted from it as a parameter and, conversely, substituted into it.

Data reuse is ensured by naming. Parameters have names, often called variables, although in functional programming they do not change values within the same context. Purely at the level of concepts, a variable is a named part of memory intended for multiple access to mutable data while a constant insures access to immutable data. In functional programming, changing the relationship between a name and a value is possible only by moving to another context, which is equivalent to changing the name. Functional variables are admissible and equal to regular constant functions and can be argument values or generated as the results of other functions. When implementing a method for the execution of a certain algorithm, the process of calculation on given arguments is often viewed as the execution of an immutable program, a predefined constant construction. In practice, in addition to such constant functions, variable functions are quite admissible. A function is a correspondence between arguments and results, both of which as well as the correspondence itself can be the values of variables. The lack of skills in working with functional variables only means that it is necessary to study this option since its potential can exceed expectations now that programming is becoming more component-oriented.

2.2. Pragmatic principles

Functional programming supports pragmatic principles for computations, such as flexibility of constraints, immutability of data, and strictness of the result. Pragmatic principles are supported by a programming system or, more precisely, by its developers.

Flexibility of constraints. On-line analysis of memory allocation and cleaning is supported to prevent unreasonable memory downtime.

Sometimes, there is enough memory for the entire task but not enough for some data blocks which may be of little importance for the task solution. In functional programming systems, such problems are solved by the principle of the flexibility of total constraints on spatial characteristics. Situations arise when some of these memory parts are exhausted, while others have underutilized space. To solve this problem, a special function is used – a "garbage collector" – which tries to automate memory reallocation or cleaning when some memory area is insufficient. This means that data may be of any size. New efficient implementations of garbage collection take into account the advantages of bottom-up processes on large amounts of memory. Many up-to-date programming systems include these mechanisms regardless of the paradigm.

Immutability of data. The representation of each result of a function is placed in a new part of free memory without distorting the arguments of this function, which can be useful for other functions.

Access to evaluated data is possible any time, which greatly simplifies program debugging and ensures the reversibility м of any actions. This ensures that all intermediate results are saved, can be analyzed and reused at any time. If the definition of a function is a static construct, the process can be viewed as a composition of functions unfolded dynamically according to this construct. A separate aspect is associated with transition from integers to real numbers, possibly requiring a change in the representation accuracy during calculations. Logically, they remain constants, but the programming system treats them as variables.

Strictness of the result. Any number of function results can be represented in a single symbolic form, from which the desired result can be selected if necessary.

This practice is convenient for describing the means of program interpretation. The boundary between arguments and results placed on the stack is always clear: the result of the function evaluated last is at the top of the stack. This principle is often interpreted as a requirement for a single value of a mathematical function, which leads to doubts about the validity of functions of integer division, root extraction, inverse trigonometric functions and many other categories of mathematical functions. This situation was considered by Fichtengolts in the first edition of a textbook on mathematical analysis.

Long before the advent of computers, ways to define and implement functions were quite varied. Approaches to storing the results of functions also differed; a common way to store data were mathematical tables or special devices, such as a slide rule. Methods for solving the same problems also vary. For example, there are more than twenty sorting methods giving the same results. The choice of a particular method depends on the conditions of program application, characteristics of the sorted data and efficiency criteria. The choice of a technique for implementing a function usually depends on how the definition of the rule and on the methods for obtaining the result according to the rule given.

They often use numbers and codes to ensure reliability and safety and to reduce the human factor. Nevertheless, many modern information services have solutions that essentially decrease their reliability and security. Password tools often provide a button to display the text. The dialogue with a "personal account" often contains a simple procedure for changing the password. User identification on the websites dealing with money and documents is done by the IP address, which ignores the fact that a computer can have many users.

2.3. Consequences

Presenting an algorithm in the form of a functional program result has important practical consequences. Constructiveness, self-applicability and factorization follow from semantic principles. Pragmatic principles lead to the hidden grammar of continuous processes, reversibility of actions and unary functions. These consequences are the basis for the intuitive construction of functional models and make it possible to carry out and understand a direct computer experiment. In addition, a number of semantic and pragmatic principles support the development of the functional models of the programs for organizing parallel computations, which can be reduced to the complexes of non-deterministic threads.

Constructiveness is a consequence of the universality principle, which allows program representations to be processed in the same way as any data.

Data representing a value or a analogy or similarity to the processed data or prototypes. Mixed and partial computations are possible, as well as optimizing transformations, macro-generation and many other tools necessary for the development for selecting a fragment to be substituted as a data part, as well as a parameter or a function definition. This provides support for meta-compilation, including syntax-driven methods of program generation and analysis. Also, uniform representations of programs are supported, externally preserving the analogy or similarity to the processed data or prototypes. Mixed and partial computations are possible, as well as optimizing transformations, macro-generation and many other tools necessary for the development of operating systems and programming systems.

Provability is based on the connection of the self-determination principle with the methods of recursion, mathematical induction and logic.

It becomes possible to deduce logically separate properties of programs and, owing to this, to detect some subtle errors. If the representation of an object is similar to some inference logic, its properties can be inferred using this logic. Thanks to factorization, it is possible to construct projections similar to a scheme admitting a proof. Most of the software verification systems are created within the framework of functional programming. This increases the reliability and security of programs, although it does not allow solving the correctness problem in full. Difficulties are associated with the insufficiency of classical logic in relation to the non-classical logic of programming.

Factorization directly follows from the principle of parameter independence taking into account the principle of universality.

Parts of data and any subformulas are the equivalents of the function parameters. For any data element with one or several selected fragments, it is possible to represent a function the parameters of which will be the fragments selected. Their substitution produces a data element equivalent to the original one, which allows using the concept of selection or partial computation.

Any marked set of program fragments can be removed from the data representing the program and associated with a certain name in order to allow the original representation to be restored. You can note that the parameters of a function call are calculated at the same level of hierarchy, in the general context, according to the principle of data immutability. Therefore, the order of evaluating the parameters can be arbitrary. Owing to this feature, we can decompose a program into autonomously developed modules, accumulate correctness, and represent parallel threads, lazy or early computations. We can say that a program can be represented in a factorized form according to various parameters, depending on the purpose of its transformation and further development. Due to the reversibility of actions, i.e., data immutability, the process of program debugging acquires convergence and allows one to bring the program to the limit of compliance with the problem statement.

The consequences of supporting the pragmatic principles in functional programming systems form intuitive images, such as process continuity (infinity), reversibility of actions, and unary functions, which provide the basis for functional models construction. In addition, a complex of semantic and pragmatic principles provides support for the development of the functional models of programs for organizing parallel computations, which can be reduced to complexes of non-deterministic threads.

Process continuity intuitively follows from the pragmatic support of the principle of the flexibility of constraints.

An execution of any function can be followed by an execution of another function. The STOP command is not a function as it has no arguments or results; it is just a signal to the processor to stop working. When executing any function, one can simultaneously execute other functions and before executing any function, other functions can be executed. This allows a significant part of work to be done on the basis of the unlimited memory model without much concern about its boundaries and the variety of characteristics of the speed of access to different data structures. Many functional programming languages support the imitation of work with infinite data structures.

Reversibility of actions is based on the illusion of data immutability, the mechanisms of which are hidden in the programming system.

After the execution of any function, you can return to the point of its call. Any function can be repeated with the same parameters; otherwise, it can be interpreted in a different way, or any other function can be executed instead. Their application requires almost no attention on the part of the programmer and debugger. The necessary data changes, such as memory reuse, are easy to automate, which allows supporting the mechanism for memorizing functions on the previously processed arguments. The programmer does not have to interfere with the implementation of such facilities as long as there are no performance problems.

Unary function is based on the principle of a rigorous result and similarly allows a function of any number of arguments to be converted to a unary function with a single argument.

For any function with an arbitrary number of parameters, you can construct its equivalent with one parameter. Since results are often arguments to enclosing functions, the accompanying principle of unary functions logically arises. In addition, the ability to proceed from a list of parameters or results to a single argument or a strict result allows replacing the usual scheme of operations (mapping two operands to one result) with mapping a set of operands to a set of results.

This set of consequences of the principles of functional programming allows refining and improving the programmed solutions when debugging the algorithms for solving new problems. It also allows for the multiple definitions of functions when the properties of the problem being solved are repeated with the same parameters; otherwise, it can be interpreted in a different way, or any other function can be executed instead. Their application requires almost no attention on the part of the programmer and debugger. The necessary data changes, such as memory reuse, are easy to automate, which allows supporting the mechanism for memorizing programs.

2.4. Applications to parallel computing

Parallelism arises from a general complex of semantic and pragmatic principles, which allows, if necessary, considering any amount of represented data and reorganizing the space of threads. These principles make functional programming convenient for working with programs aimed at parallel processes organization. The first is the principle of the equal rights of parameters, which guarantees the same context when the parameters of a function of the same level are evaluated. It becomes possible to represent independent threads and combine them into multithreaded or multiprocessor programs and into a common problem-oriented complex. In addition, parallelism uses the principles of a strict result and universality, which allows us, if necessary, to consider any number of represented threads and to reorganize the thread space. The pure functional programming principles are not adequate to model the interacting and imperatively synchronized processes. Parallelism clarifies these principles, which facilitates the development of parallel computing programs.

Repression of small probabilities is aimed at preventing an excessive number of threads corresponding to highly unlikely situations, which somewhat narrows the principle of universality.

The principle of universality has two aspects – equal rights of programs and data and completeness of function definitions. When solving the problems of parallel computing, universality preserves the equality of programs and data, which is traditionally in demand in the tasks of operating systems. The completeness of functions, convenient for building programs from the modules that have already been debugged, can create problems because the number of threads in multiprocessor programs associated with rare diagnostic situations increases. Any fragment which is unlikely or impossible to calculate can be removed from the function representation and converted into a delayed action. Branches for practically irrelevant situations can be deleted or moved to the debug version. Sometimes, this problem can be overcome by choosing the expressions that do not require branching, but more often it is done by checking data types. The amount of necessary diagnostics can be partially reduced by means of the static analysis of data types.

Load balancing reduces the real runtime of a program, which can be viewed as a transfer of the principle of flexibility to time constraints.

Lazy, or early, computations provide the ability to redistribute the load quickly. A complex function definition can sometimes be reduced to two functions, the first of which performs a part of the definition, postponing the execution of the rest, and the second resumes the execution of the delayed analogy or similarity to the processed data or prototypes. Mixed and partial computations are possible, as well as optimizing transformations, macro-generation and many other tools necessary for the development leading to a dangerous stack growth. It should be noted here that many functional programming systems offer a number of practical solutions. These include delayed actions, memoization, ascending recursion, dynamic programming techniques, and optimization of recursions by reducing to loops, which in many cases makes it possible to eliminate an excessive stack swelling. Operations with a stack within the framework of the principle of flexibility of constraints can be supported more efficiently than in most programming languages and systems. In addition, the factorization of programs into schemes and fragments allows components separation according to the level of debugging complexity and inheriting the correctness of the modules previously debugged. The functions used do not require a preliminary evaluation of parameters, like macro technology. Similarly, in the mpC language, the amount of computation is redistributed when an unbalanced load of processors analogy or similarity to the processed data or prototypes. Mixed and partial computations are possible, as well as optimizing transformations, macro-generation and many other tools necessary for the development parallelization.

Any finite set can act as an iteration space for a function defined on it. If there is a set of data such that the calculation of a function on one of its elements does not require its results for other data, then it is convenient to use this set as an iteration space for the simultaneous execution of this function on all of its elements. The equal rights (independence from each other) of parameters is becoming increasingly important; it provides a solution to the problems of thread reorganization, when multiprocessor systems requiring the decomposition of program fragments are structured for different configurations.

The technique and concept of iterating spaces is convincingly supported in the Sisal language, in which iteration spaces are constructed over enumerable sets using scalar and Cartesian product operations [33].

Parallelism uses the principles of equal rights for parameters and strict result, allowing, if necessary, any number of parameters or results to be considered as their common data structure. This makes it possible to represent independent threads and to combine multi-threaded or multiprocessor programs into a single complex. In addition, note that the parameters of a function call are calculated at the same level of hierarchy, according to the principle of data immutability, in the general context. Therefore, the order of parameter calculation does not matter; it can be arbitrary. This makes functional programming convenient for working with programs intended for parallel processes organization.

The solution is somewhat more complicated when pragmatic principles requiring a revision of system solutions at the level of programming system development are involved.

Automatic parallelization consists in extracting from the program the autonomous parts allowing independent execution. Suppose there is a function F with a known execution time T, which can be decomposed into two functions, F1 and F2, such that the execution time of each of them is noticeably less than T. Then, if they are independent, they can be executed in an arbitrary order, and the execution time of the original function will be less than T.

Identity of repeated runs for the purpose of program debugging and performance measurement.

The transition to supercomputers has shown that, with too many processors, there is no longer possibility to observe program re-execution necessary for debugging and measurements. During the next run, there may be failures on different processors. Here, functional programming can allow special interpretations of the program by taking into account the protocols and results of the previously executed runs with tracking the execution identity.

Multi-pin fragments, such as control schemes or operations producing more than one result, at first glance contradict the principle of a strict result.

However, the possibility of transition from a strict result allows one to build multi-threaded functions taking parameters from a set of peer-to-peer threads and generating a set of results in terms of the number of threads. Thus, it is possible, as in the functional parallel programming language Sisal, to switch to the operations mapping a string of operands to a string of results [33]. This may correspond to the structure of some hardware nodes and thus allows presenting more efficient solutions.

2.5. Performance increase

In the transition to reusable programs and parallel computing, application success and program performance become more important than their formal correctness and efficiency. Pure functional programming can be viewed as a functional modeling technique for prototyping complex problem solving programs. A broader paradigm of functional programming applied in production allows one to move from such functional models and prototypes to more efficient data structures, making practical decisions and trade-offs in their processing depending on real conditions, when necessary. In addition to the principles and their consequences in real programming languages and systems, the production paradigm of functional programming allows the inclusion of balancing mechanisms, which look like special functions in a programming language. For example, Lisp 1.5, Clisp, Cmucl, and other members of the Lisp family typically provide the following trade-off functions:

- ***data type control*** softens the principle of universality by the functions of static and dynamic analysis of data types;
- ***data recovery*** makes it possible to eliminate excessive memory consumption, partly counteracting the principle of data immutability when destructive functions having safe analogs are used;
- ***loops schemes***, simulating a slightly expanded variety of familiar control mechanisms for computations, overcome typical concerns about the implementation complexity of the principle of self-determination;
- ***accounting for predictions*** on memory size and execution speed allows us to program memory allocation in order to neutralize inaccurate flexibility constraints;
- ***pseudo-functions***, in addition to result generation, act upon external memory or interact with devices including the fulfillment of I / O and file operations, provided by the principle of parameter independence;
- ***memoization*** allows a radical reduction in the complexity of repeated calculations by storing the results for all parameter values, which expands the principle of the strictness of the result. The results of all threads have equal rights and all of them can be saved and reused without unnecessary calculations.

Table 2 presents the general interaction of principles, consequences, applications and trade-offs in functional programming systems.

Table 2.
Key aspects of the relationship between semantic and pragmatic mechanisms

	Semantics	Pragmatics
Principles	universality self-definition parameter independence	flexibility of constraints data immutability strictness of the result
Consequences	constructiveness provability factorization	continuity of processes reversibility of actions unary functions
Applicatio ns to parallel computing	repression of small probabilities load balancing iteration spaces	automatic parallelization Identity of repeated runs multi-pin fragments
Practical trade-off	data type control loops schemes recovery of data	programmable accounting for predictions pseudo-functions memoization

Within the framework of functional programming, it is possible to take into account the specifics of parallel computations affecting the choice of methods for their solution, depending on the priorities in the choice of language means and implementation possibilities.

Paradigmatic errors found in running highly popular programming systems on modern multiprocessor and network hardware show that some of them were invisible before the advent of networks, mobile devices, and supercomputers.

Consequences of the semantic and pragmatic principles of functional programming and high modeling power of the apparatus of functions, extended with special functions of practical compromises, make it possible to supplement efficiently the main paradigms of parallel computing and practical work on program performance improvement.

3. Conclusion

In February 2021, the 22nd conference on modern trends in functional programming was held [1]. The reports presented convincingly showed that the focus of functional programming was on solving the problems of parallel computations organization.

Functional programming makes it possible to take into account both the aspects of the problems solved and programming methods necessary to solve parallel computation problems. It is possible to form a sequence of comparable program examples, allowing a comparison of different programming languages.

It is possible to analyze the results of the direct measurement of program performance and to highlight the features of the basic tools and implementation solutions in programming systems designed to improve the software products developed. The errors caused by the choice of paradigms found when operating familiar programming systems on modern multi-processor equipment show that some of them were simply invisible before the advent of networks, mobile devices and supercomputers.

Some questions are yet to be answered in practical terms. The emergence of new paradigms can be associated with new problems, which are still difficult to solve. It is not clear how appropriate the interaction of paradigms is. The educational problems of learning new paradigms are left aside. The educational programs of many universities include teaching parallel computing, which is sufficient to understand the complexity of new paradigms and set the appropriate research objectives. However, university students lack practice in parallel computing.

The problem is in the implementation of high-performance programs satisfying the complicated and difficult to verify criteria of reliability and safety. A characteristic feature of the functional approach as a programming method is the transition to the classes of problems in the process of a meaningful analysis of the formulation of any problem. When solving problems, class boundaries are established. Experiments on supercomputers have shown that system solutions can contribute significantly to parallel computing performance, and this contribution may even exceed the theoretical forecasts. This justifies the need for a more fundamental approach to programming, especially to system programming and its mathematical foundations, naturally represented in the purely functional programming [14, 16].

When creating, forming and investigating the mathematical models as the fundamental basis for solving the difficult problems of software efficiency, reliability and safety, it is important to develop the models related to time and resources, which are poorly presented in classical mathematics curricula. However, they are available in functional programming languages and systems. Extensive development of IT noticeably outstrips human capabilities to learn new options of IT hardware and tools, which go beyond the user level supported by the suppliers. The mission of programming is to create the tools intended to improve the quality of information systems, including search for new solutions to the problem of reliability and security of information technologies [34, 35].

In addition, paradigm features are only partially expressed at the level of program representation. Other requirements are expressed at the level of pragmatics or implementation of the programming language.

Discipline** of **access to multilevel heterogeneous memory and protocols for interaction between processes is examples of a problems that has not yet received a convenient solution. Here it is required to refine the mechanisms of data immutability, possibly in favor of their recoverability.

Data immutability is preserved at the local thread level, but it causes problems when moving to shared memory. There are data blocks that are different in size, time of access to them and in the discipline of service, possibly available simultaneously to different functions and to data storage . If data blocks are available to different functions, then they can act as protocols, messages, and other representations of dependencies between functions. Usually such dependencies are represented in shared memory. Possibly, shared memory mechanisms require data recovery, apart from the mathematical aspects of working with heterogeneous memory, copies, replicas, etc., which is similar to dynamic editing of complex structures already illustrated in the problems of working with the DSL languages [1].

The A.P. Ershov Institute of Informatics Systems SB RAS has been traditionally engaged in the creation of educational programming languages including familiarization with the phenomena of parallelism. Currently, the multi-paradigm language SINHRO is being developed [8]. It is desirable that a modern language for the development and debugging of long-lived parallel computing programs include sublanguages intended to support the main parallel programming paradigms, inheriting the experience of the languages previously created.

References

[1] Koopman P., Michels S., Plasmeijer R.. Dynamic editors for well – typed expressions // Trends in Functional programming, 22nd International Symposium TFP 2021, February 17–19, 2021. – 2021. – P. 44–66. – (Lect. Notes in Comp. Sci.; 12834).

[2] Gorodnyaya L.V., Marchuk A.G. Development of parallelism models in high level languages (Razvitiye modeley parallelizma v yazykakh vysokogo urovnya) // Nauchnyy servis v seti Internet: vse grani parallelizma. – P. 342 – 346. – http://agora.guru.ru/abrau2013/pdf/342.pdf (In Russian).

[3] Gorodnyaya, L. On parallel programming paradigms // Proc. CEUR Workshop 2015. – 2015. – No.1482. – P. 587–593.

[4] Gorodnyaya L.V. On the implicit multi-paradigmality of parallel programming (O neyavnoy mul'tiparadigmal'nosti parallel'nogo programmirovaniya) // Proc. Nauchnyy servis v seti Internet: XXIII Vserossiyskoy nauchnoy konferentsii, 20 – 23 September 2021. – M.: IPM im. M.V. Keldysha, 2021. – P. 104 – 116. – https://doi.org/10.20948/abrau – 2021 – 6, https://keldysh.ru/abrau/2021/theses/6.pdf (In Russian).

[5] Kotov V.Ye. MARS: architectures and languages for parallelism implementation (MARS: arkhitektury i yazyki dlya realizatsii parallelizma) // Sistemnaya informatika. Iss 1. Problemy sovremenogo programmirovaniya. – Novosibirsk: Nauka, 1991. – P. 174 – 194 (In Russian).

[6] Dushkin R.V. Functional Programming in the Haskell language (Funktsional'noye programmirovaniye na yazyke Haskell). – M.: DMK – Press, 2016 (In Russian) .

[7] Gorodnyaya L., Shilov N. Educational value of teaching parallel programming paradigm // Proc. PSI ’11, School Informatics. – 2011. – P. 1 – 6.

[8] Gorodnyaya L.V. The SINHRO parallel programming teaching language (Uchebnyj yazyk parallel'nogo programmirovaniya SINHRO) // Proc. Yazyki programmirovaniya i kompilyatory – 2017 / Ed. D.V. Dubrov. – Rostov-on Don: Izd. Yuzhnogo federal'nogo universiteta, 2017. – P. 92–97. – http://plc.sfedu.ru/files/PLC – 2017 – proceedings.pdf (In Russian) .

[9] Shilov N.V., Gorodnyaya L.V., Marchuk A.G. Parallel programming amid other programming paradigms (Parallel'noye programmirovaniye sredi drugikh paradigm programmirovaniya) // Prikladnaya informatika. – 2011. – No.1 (31). – P.120 – 129. –_http://agora.guru.ru/abrau2011/pdf/193.pdf (In Russian).

[10] McCarthy J. LISP 1.5 Programming Manual. – Cambridge: The MIT Press, 1963.

[11] Backus J. Can programming be liberated from the John von Neumann style? A functional style and its algebra of programs // Commun. ACM. – 1978. – Vol. 21, Iss.8. – P. 613 – 641.

[12] Khenderson P. Functional Programming (Funktsional'noye programmirovaniye). – M .: Mir, 1983 (In Russian).

[13] H'yuvenen E., Seppanen Y. The Lisp World (Mir Lispa). – M.: Nauka, 1994 (In Russian).

[14] Lavrov S.S. Functional programming (Funktsional'noye programmirovaniye) // Komp'yuternyye instrumenty v obrazovanii. – 2002. – No. 2 – 4 (In Russian) .

[15] Lavrov S. S., Gorodnyaya L.V. Functional programming. The Lisp language interpreter (Funktsional'noye programmirovaniye. Interpretator yazyka Lisp) // Komp'yuternyye instrumenty v obrazovanii. – SPb, 2002. – No.5 (In Russian) .

[16] Gorodnyaya L.V. Functional Programming Basics. Series of lectures. (Osnovy funktsional'nogo programmirovaniya. Kurs lektsiy). Uchebnoye posobiye. Seriya «Osnovy informatsionnykh tekhnologiy». – M.: INTUIT.RU, 2004. – http://www.intuit.ru/studies/courses/29/29/info (In Russian).

[17] Gorodnyaya L.V., Berezin N.A. Introduction to Lisp Programming (Vvedeniye v programmirovaniye na Lispe). – M.: Internet – Universitet

Informatsionnykh tekhnologiy, 2007. – http://www.intuit.ru, http://www.intuit.ru/studies/courses/1026/158/info (In Russian).

[18] Gorodnyaya L.V. The first implementations of the Lisp language in the USSR (Pervyye realizatsii yazyka Lisp v SSSR) // Proc. SoRuCom – 2011. – P. 95– 100. – https://www.computer – museum.ru/histsoft/lisp_sorucom_2011.htm (In Russian).

[19] Gorodnyaya L.V. Implementation of the Lisp interpreter (Realizatsiya Lisp – interpretatora). − VTS SO RAN SSSR, Novosibirsk, 1974. − P. 24 – 35 (In Russian).

[20] Lavrov S. S. Methods to define the semantics of programming languages (Metody zadaniya semantiki yazykov programmirovaniya) // Programmirovaniye. – 1978. – No. 6. – P. 3 – 10 (In Russian).

[21] Gorodnyaya L.V. On the presentation of the results of the analysis of programming languages and systems (O predstavlenii rezul'tatov analiza yazykov i sistem programmirovaniya) // Proc. Nauchnyj servis v seti Internet: XX Vserossijskaya nauchnaya konferencia, 17 – 22 sentyabrya 2018, Novorossiysk. – M.: IPM im. M.V. Keldysha, 2018. – P. 262–277. – https://doi.org/10.20948/abrau – 2019 – 03 (In Russian).

[22] Gorodnyaya L. On the presentation of the results of the analysis of programming languages and systems // Proc. CEUR Workshop. – 2018. – Vol. 2260. – P. 152–166.

[23] Gorodnyaya L. V. Systematization of programming paradigms based on decision making priorities (Sistematizatsii paradigm programmirovaniya po prioritetam prinyatiya resheniy) // Elektronnyye biblioteki. – 2020. – Vol. 23 –.No.4. – P. 666 – 696. – https://doi.org/10.26907/1562 – 5419 – 2020 – 23 – 4 – 666 – 696 (In Russian).

[24] Gorodnyaya L. Method of paradigmatic analysis of programming languages and systems // Proc. CEUR Workshop. – 2020. – Vol. 2543. – P. 149 – 158.

[25] Wegner P. Concepts and paradigms of object-oriented programming // SIGPLAN OOPS Mess. – 1990. – Vol.1. – No.1. – P. 7 – 87. – https://pdfs.semanticscholar. DOI: http://dx.doi.org/10.1145./

[26] Gorodnyaya L. V. Programming paradigm: Textbook (Paradigma programmirovaniya: uchebnoye posobiye). – Sankt-Peterburg: Lan', 2019. – https://e.lanbook.com/book/118647 (data obrashcheniya: 12.11.2021) (In Russian).

[27] Gorodnyaya L.V. Programming Paradigms (Paradigmy programmirovaniya). – M.: INTUIT.RU, 2007. – https://intuit.ru/studies/courses/1109/204/info (In Russian).

[28] Gorodnyaya, L. V. Programming Paradigm: Series of Lectures (Paradigma programmirovaniya: kurs lektsiy). – Novosibirsk: RITS NGU, 2015 (In Russian).

[29] Gorodnyaya L.V. Programming Paradigms: Analysis and Comparison (Paradigmy programmirovaniya: analiz i sravneniye). – Novosibirsk: Izd. SO RAN, 2017 (In Russian).

[30] Gorodnyaya L.V., Kirpotina I.A. On the problem of the reliability of the historical factography available on the internet (O probleme dostovernosti dostupnoy v Internete istoricheskoy faktografii) // Proc. SoRuCom – 2017, Zelenograd, 3–5 October 2017. – Moskva: FGBOU VO “REU im. G.V.Plekhanova”. – 2017. – P. 40 – 49 (In Russian).

[31] Kernigan, B.W., Ritchi, D.M. The C Programming Language (YAzyk programmirovaniya Si). – Moskva: Finansy i statistika, 1992 (In Russian).

[32] Lastovetsky A.L. mpC: A Multi – Paradigm Programming Language for Massively Parallel Computers // ACM SIGPLAN Notices. – 1996. – Vol. 31. – No. 2. – P. 13–20.

[33] Kasyanov V.N. Sisal 3.2: functional language for scientific parallel programming // Enterprise Information Systems. – 2013. – Vol. 7. – No. 2. – P. 227–236.

[34] Nahhas S., Bamasag O., Khemakhem M. and Bajnaid N. Added Values of Linked Data in Education: A Survey and Roadmap // Computers. – 2018. – No. 7. – https://doi.org/10.3390/computers7030045.

[35] Gorodnyaya L.V. The strategically promising programming paradigms of Academician Andrey P. Ershov (Perspektivno strategicheskie paradigmy programmirovaniya Akademika Andreya Petrovicha Ershova) // Proc. SORUCOM, Moskva, 6–8 October 2020. – 2020. – P. 83–97 (In Russian).

[36] Gorodnyaya L. Strategic paradigms of programming initiated and supported by Academician Andrey Petrovich Ershov // Proc. SoRuCom 2020. – 2020. – P. 1–11, 946 – 972.