ISSN 0146-4116, Automatic Control and Computer Sciences, 2021, Vol. 55, No. 7, pp. 636–646. © Allerton Press, Inc., 2021.
Russian Text © The Author(s), 2020, published in Modelirovanie i Analiz Informatsionnykh Sistem, 2020, No. 2, pp. 164–179.

# Dynamically Changing Parallelism with Asynchronous Sequential Data Flows

**A. I. Legalov[a], * (ORCID: 0000-0002-5487-0699),**
**I. V. Matkovskii[a], ** (ORCID: 0000-0002-4801-7982),**
**M. S. Ushakova[a], *** (ORCID: 0000-0003-4234-2714),**
**and D. S. Romanova[a], **** (ORCID: 0000-0002-9020-4802)**

[a] *Siberian Federal University, Krasnoyarsk, 660041 Russia*

**e-mail: legalov@mail.ru*

***e-mail: alpha900i@mail.ru*

****e-mail: ksv@akadem.ru*

*****e-mail: daryaooo@mail.ru*

Received May 27, 2020; revised June 8, 2020; accepted June 10, 2020

**Abstract**—A statically typed version of the data driven functional parallel computing model is proposed. It enables a representation of dynamically changing parallelism by means of asynchronous sequential data flows. We consider the features of the syntax and semantics of the statically typed data driven functional parallel programming language Smile, that supports asynchronous sequential flows. Our main idea is to apply the Hoar concept of communicating sequential processes to the computation control on data readiness. It is assumed that on data readiness a control signal is emitted to inform the processes about the occurrence of certain events. The special feature of our approach is that the model is extended with special asynchronous containers that can generate events on their partial filling. These containers are a stream and a swarm, each of which has its own specifics. A stream is used to process data which have identical type. The data comes sequentially and asynchronously at arbitrary time moments. The number of the coming data elements is initially unknown, so the processing completes on the signal of the end of the stream. A swarm is used to contain independent data of same type and may be used for massive parallel operations performing. Unlike a stream, swarm's size is fixed and known in advance. General principles of operations with asynchronous sequential flows with arbitrary order of data coming are described. The use of streams and swarms in various situations is considered. We propose language constructs which allow operating swarms and streams and describing the specifics of their application. We provide sample functions to illustrate the use of different approaches to describing the parallelism: recursive processing of asynchronous flows, processing of flows in an arbitrary or predefined order of operations, direct access and access by reference to elements of streams and swarms, pipelining of calculations. We give a preliminary parallelism assessment which depends on the ratio of the rates of data coming and their processing. The proposed methods can be used in the development of the future languages and toolkits of architecture-independent parallel programming.

**Keywords:** parallel computations, asynchronous computations, static typing, dynamically changing parallelism

**DOI:** 10.3103/S0146411621070105

## INTRODUCTION

There are two main approaches in the development of architecture-independent parallel programs:

• to develop and debug sequential programs independently of parallel computing systems (PCSs) and then make them parallel to suit the target architecture;

• to develop programs or algorithms that right away describe the maximum parallelism of the problem to solve, and then to "compress" this parallelism to the target architecture.

In practice an intermediate solution is often used, when a parallel program is initially developed taking into account the peculiarities of the target architecture. However, unlike the first two approaches, the resulting code turns out to be strongly dependent on the target architecture, which makes it difficult to transfer it to other PCSs.

At the same time, regardless of the used approaches, it should be noted that almost in any of these three approaches, program parallelism is bound by the initially specified basis of operations and also by the methods of parallel processes describing. Usually, this is due to the computation models, linguistic and instrumental tools built on their basis rather than to the architectural dependence.

Bounding the parallelism from above (maximum parallelism) as well as from below (sequential execution) leads to a semantic break at the level of a real computing. That is, executing a program in real computing resources results in, the loss of efficiency and balance due to the fact that the characteristics of the parallel algorithm, that were fixed during the development, do not coincide with the dynamic characteristics of the PCS subsystems. That is why maximum parallelism is often compressed when fitting the program to the target architecture, which consists in solving the problem inverse to the parallelization of sequential programs. This, in turn, leads to a loss of efficiency in the parallel software development process and does not allow writing a single program suitable for different parallel architectures.

Hence, it is topical to develop parallel computing models, language, and tools that would allow us to fit a once developed computer program to various architectures in order to effectively exploit computational resources.

One such solution is to use data as soon as they become available, without accumulation data in arrays before subsequent calculations. It is assumed that data processing starts as soon as individual elements are ready. The parallelism in the processing of such data depends both on the rate at which these elements are placed in the arrays and on the time of their subsequent processing. The ratio between these rates allows us to consider different levels of parallelism of the corresponding algorithms from sequential computations to unlimited parallelization. One such data structure, an asynchronous list, was proposed as part of an extension of the data driven functional parallel computing model (DDFPCM) [1].

At the same time, it should be pointed out that this type of asynchronous data flows has the following disadvantage: the order in which arguments are received for processing may not correspond to their order in the output stream, that is used to collect the results. This transposition may not be critical for some algorithms. However, in most cases, this situation is unacceptable. One solution to this problem is to organize the ordered queues, where the result elements are coming in the same order as the corresponding coming arguments. However, in this situation, there appears an implicit synchronization of the stream of asynchronously coming data, which, in turn, leads to slower processing.

To solve this problem, we propose an approach based on extending the capabilities of the parallel list of the DDFPCM. It is assumed that data elements coming in this list are immediately passed to processing, after which the result elements are placed in the same positions in the result list as the corresponding input elements. An additional semantic change can also be made to the previously discussed asynchronous list. The application of the proposed changes allows us to create a new kind of computation model describing asynchronous algorithms with dynamically variable parallelism.

In the current paper we consider the features of semantics of the operators that provide a support for asynchronous data flow computations, as well as their mapping to the syntax and semantics of the statically typed data driven functional parallel programming language Smile. We give some examples to demonstrate the specifics of the proposed structures. We perform a preliminary estimation of parallelism depending on the time ratio between the rate of data coming and the rate of their processing.

## 1. KEY IDEAS OF THE APPROACH

The main idea of the approach is based on the concept proposed by Hoar [2], in which parallelism is described as the communication of the processes spawn by means of sequentially generated events. These events are sequential due to the fact that the moments of their occurrence are considered instantaneous and can be ordered. This allows ignoring concurrency. It is assumed that, instead of being simultaneous, events are nondeterministic, which means that two competing events can occur in an arbitrary sequence. A similar approach is also used in a number of simulation systems based on Petri nets [3].

When control is based on data readiness, the generation of such events takes place at the moments of data occurrence. Events separated from data can be viewed as a flow of control signals, whose interaction allows forming various strategies for computations control [4, 5]. On the basis of this approach, we propose a model of an event processor [6] within the concept of data driven functional parallel programming, which traverses the data flow graph of a program and performs data processing based on responding only to control signals.

A sequence of asynchronous signals allows describing the parallelism without explicit parallelization schemes. The asynchronous lists proposed in DDFPCM [1] allow describing computations through sequences of the recursive calls. It is also shown that the use of asynchronous lists in this case provides a

dynamic change in parallelism depending on the relationship between the rate of data coming and the execution time of data processing operations, varying from maximum parallelism to sequential computations. That is, the focus on asynchronously coming data and sequences of signals, that inform about data coming, provides a more flexible description of parallel algorithms and their further adaptation to specific conditions using only one method of describing the algorithm based on sequentially processed asynchronous data flows. The disadvantage of asynchronous lists associated with the nondeterminism of their data coming is proposed to be corrected by modifying parallel lists and their extension with functions that support additional control mechanisms based on data readiness signals.

The use of event-driven computation control in systems based on data-readiness control was proposed in [6]. Within this concept, an interpreter of data driven functional programming language Pifagor was developed [7]. Event control allows separating the control flow graph from the data flow graph. At the same time, it gave an opportunity to change the computation control strategy by changing the control flow graph without changing the data flow graph of the program.

Further development of the presented investigations is devoted to the extension of the semantics of the DDFPCM and the introduction of static data typing into the model and the language, which provides a more flexible transformation into other parallel architectures. These changes result in the creation of a statically typed model of data driven functional parallel computations (STMDDFPC) and the development of a statically typed data driven functional parallel programming language Smile [8] based on this model.

## 2. KEY CONCEPTS OF THE COMPUTING MODEL SUPPORTING ASYNCHRONOUS SEQUENTIAL DATA FLOWS

New proposed concepts extend the capabilities of some program-forming operators of DDFPCM. However, the use of static data typing instead of dynamic typing in the new model and programming language, on the one hand, imposes some restrictions, but, on the other hand, provides additional opportunities for transforming data driven functional parallel programs into programs for real architectures.

In the STMDDFPC, in comparison with DDFPCM, the semantics of container data types has been changed, which provide support for massive operations. In particular, to support compile-time type checking, a vector is used instead of the data list. All its elements are of the same predefined (named) type. Elements of a swarm as well as of a stream are also of the same type. This makes it possible to introduce massive element-wise operations on these containers, as well as, regardless of the container type, to apply functions that process the container as a whole.

The use of static typing also led to the separation of the interpretation operator into two different types: single (single-argument) and group (massive, element-wise) operator. In a textual representation the single operator of interpretation is denoted (as before [9]) by ":" (postfix form) or "^" (prefix form). It is used to apply ordinary functions that take the argument as a whole. The massive operator of interpretation is used to perform calculations on each element of the same type in the container, generating at the output a container with elements of the type corresponding to the type of the result of the function being applied. It is denoted by the double symbol "::" for postfix or "^^" for prefix forms respectively.

Introducing two different notations allows us to use a function with the same name in different contexts unambiguously. For example, the subtraction function "-" applied to the argument (10, -3), taken as a vector consisting of two integers, produces the following values:

```
// binary subtraction function over argument as a whole
(10, -3):- => 13
// sign changing function is massively applied
// to two arguments of the same type
(10, -3)::- => (-10, 3)
```

The separation of the interpretation operator into massive and single-argument ones allows us to introduce a set of additional more flexible one-argument functions for the stream and the swarm, which provide processing of asynchronously coming data.

### *2.1. Organization of Asynchronous Sequential Streams with Arbitrary Order of Data Coming*

The concept *stream* introduced in the STMDDFPC extends the concept of the previously proposed asynchronous list. The basic idea remains the same and is in asynchronous data coming. However, all elements are assumed to be of the same named type, which in turn cannot be a stream or a swarm. This is

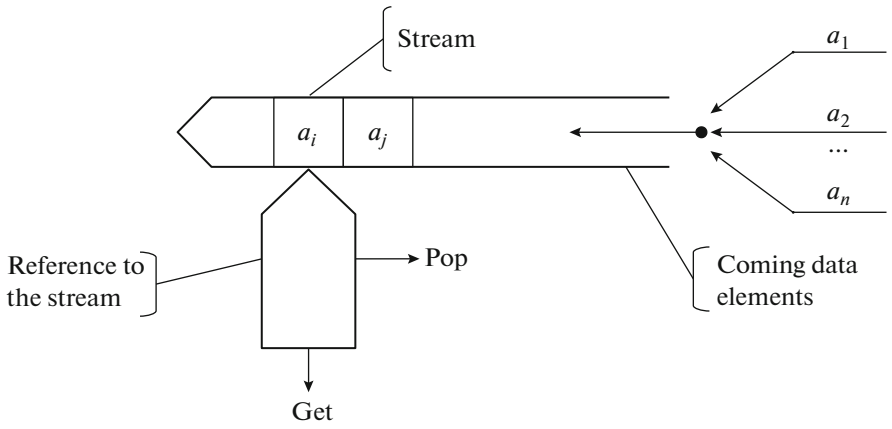


**Fig. 1.** General scheme of a stream.

consistent with the concepts of generic statically typed languages. A stream can be viewed as an entity (Fig. 1) with the following properties:

• When at least one data element is ready and is placed into a stream, it generates a signal informing about its readiness.

• Any ready data element can be read from a stream for further processing.

• If new data elements come to a stream during the processing of another element selected from the stream, they can also be selected from the stream asynchronously in the order of their coming and processed in parallel.

• If two elements of a stream are processed in parallel, then the results of the processing may be placed into another stream of the corresponding type; the order of the results in the latter stream depends on the processing time and may differ from the initial order of the corresponding arguments.

• A stream can be checked for availability of further coming data, which allows completing the processing.

**2.1.1. Basic stream operations.** A stream in the Smile programming language is defined by the following syntax:

```
stream = element_type_name "{}".
```

Operations with streams can be performed using the massive operator of interpretation. In this case, all elements of the processed stream, after applying a function, come to the generated output stream, that accumulates the results of calculations. For example, the application of the function `sin` to all elements of the input stream X with the output stream Y in the Smile language can be written as follows:

```
X::sin >> Y
```

where the stream X is previously described as `X@float{}`. The `@` character separates the name of the used entity from its type. The output stream is automatically generated and has the same type as the result type of the function `sin`, the type is defined by function signature:

```
sin << func float -> float
```

At the same time, the order of result elements in the stream denoted by Y may differ from the order in which the arguments come to the stream X.

It is assumed that operations on the stream are not applied directly. This is due to the fact that direct access to the stream can lead to side effects that change its state, which does not allow other operations to be correctly applied to the stream in parallel. Instead, the single-argument operator of interpretation operates with the stream through a reference. The presence of several references to one stream, which in fact play the role of iterators, allows processing it in different ways in different parts of the program. The syntax of a reference to a stream in Smile is as follows:

```
stream_reference = element_type_name "{*}".
```

Streams can be passed to functions as parameters by references. The function calculating the sine for all elements of the stream in Smile is as follows:

```
sinStream << funcdef X@float {*} -> float {*} {
    X::sin:return
}
```

That is, passing streams by reference allows massive function application. It should also be noted that, when a stream is used as an argument to a massive operator of interpretation, a hidden local reference to the stream is automatically created in order to avoid side effects.

In most cases, using massive stream operations is not sufficient for flexible asynchronous programming in the case when the element-wise processing or merging of data from multiple streams is needed. For this situations it is necessary to introduce additional constructs, in many respects similar to iterators. These constructs are defined by special one-argument functions of a stream. In particular, before direct access to a stream element, it is essential to check whether the stream is able to generate elements (analogous to the end-of-file condition). This is due to the fact that the number of elements coming into the stream may not be known in advance. To check the fact that the stream is able to generate data elements, we use the function `is`, which has the following signature:

```
is << func any{*} -> bool
```

`any` being a keyword denoting an arbitrary type. The function returns `true` if the stream is able to generate data elements or they are already in the stream. Otherwise, `false` is returned.

To fetch data from a stream, the function `get` is used, which reads the first element of the stream queue. If no element has been generated, the function `get` waits for its generation. If there are multiple elements in the stream queue, only one is selected. Attempting to execute this function for the stream which has already finished elements generation leads to error and interruption of the function. The function `get` has the following signature:

```
get << func any{*} -> any
```

Before reading the next element from the stream, it is necessary to remove the element that has been already read from the reference. The function `pop` is used for this. Attempting to execute this function for the stream which has already finished elements generation leads to error and interruption of the function. The function has the following signature:

```
pop << func any{*} -> any{*}
```

That is, this function returns a new reference to the same stream, but it is without the processed element (this element is no longer available through the returned reference).

As an example of using single-argument functions consider finding the sum of the elements coming into the stream:

```
sum << func X@float{*} -> float {
    if << X:is;
    if^({(X:get, X:pop:sum):+}, 0):return
}
```

The test `X:is` returns the boolean value `true/false`, which is used as a selector by the interpreter operator. If `true` is returned, the first element of the tuple is selected, triggering the left recursion for the sum function. The `false` value is generated when the stream is finished. In this case, the value 0 is returned. When the recursive call returns, the elements are summed up.

**2.1.2. Filling the stream with data from various sources.** In the above example of summing the elements of the stream, a sequential recursion is, in fact, implemented. Indeed, each return value of a recursive call is the sum of the next element and the previous intermediate accumulated return value. In article [1] it is shown that sequential recursive calls allow using an asynchronous list to implement the summation whith parallelism, which can reach maximum, that is equivalent to cascade folding, depending on the relationship between the rate of data elements coming and the rate of their processing. Using streams in the Smile programming language allows reaching the same flexibility. At the same time, the capability to create storages allows inserting not only the initial data, but also intermediate results into the stream. The corresponding function for summing the values coming into the stream is as follows:

```
// Asynchronous summation of stream elements
// with insertion of intermediate computation results
sum << func X@float{*} -> float {
    // Checking that the stream is able to generate data
    notEmpty << X:is;
    notEmpty^(
        // If at least one element is available
        {block{
            // The element is selected from the stream
```

```
            a << X:get;
            // A reference to the next position is formed
            Y << X:pop;
            // and the check for the presence of the next element is done
            notEmptySecond << Y:is;
            notEmptySecond^(
                {block{
                  // If another element is available, it can be added to the first
                  // one and sent to the stream by any available reference
                  (a, Y:get):+ -> Y;
                  // Create a new reference without the second element
                  // and continue calculations recursively
                  Y:pop:sum:break
                },
                // Otherwise, there is only one element in the stream.
                // Then, its value is the resulting sum
                a
          ):break
      }},
      // If there is no data, 0 is returned
      0.0
    ):return
  }
```

If the stream contains any two elements then the function sums them up and sends the result by reference back to the same stream. The process is recursively repeated for the remaining elements in the stream, which may be newly coming elements as well as the intermediate sum values. When the stream finally contains only one element, this element is the resulting sum.

**2.1.3. Non-deterministic behavior of a stream during asynchronous computations.** Using of streams allows organizing asynchronous computations with dynamically changing parallelism, depending on the ratio between the rate of data elements coming to the stream and the rate of their processing by functions on the stream. However, the high probability that the initial order of the input arguments will not coincide with the order of the results at the output does not allow in many cases organizing deterministic and predictable calculations. As an example, we can consider the computation of a data array coming from the input stream and, after processing, being sent to the output stream. Let the function calculate the following formula:

```
y[i] = 1 - sin(x[i])*sin(x[i]).
```

When we use streams as an intermediate storage for results, pipeline calculations are organized without any difficulties (with appropriate timing ratios). However, due to the possibility of different processing time for different stream elements, the correct sequences of values in the resulting stream may be tangled.

This situation can be illustrated by the following code in Smile:

```
Dif1Sin2Stream << func X@float{*} -> float{*} {
    result@float{};
    (X, result):GetStreamResult >> ok;
    result:ok:return
}
```

where

```
GetStreamResult << func (arg@float{*}, result@float{*})->signal {
  // Checking the stream for data availability
  if << arg:is;
  if^(
     // Writing the result to the output stream
     // after adding data elements to it
```

```
      {block {
        x << arg:get; // Getting element from the stream
        s << x:sin; Sin2 << (s,s):*; // Sine squared
        // Calculating the current value and passing it to the output stream
        (1,Sin2):- -> result;
        // The processed element is removed from the stream
        // and we switch to the next element
        (arg:pop, result):GetStreamResult:break}
      },
      // If no more data is available,
      // then return the signal corresponding to no operation.
      !
  ):return
}
```

The main function `Dif1Sin2Stream` receives data from the input stream through the reference X. The result of calculations is returned from the function via a stream reference. The stream is defined inside the function by the storage result, and the accumulation of calculation results is carried out in the function `GetStreamResult`, to which the stream is passed as a parameter.

The function `GetStreamResult` performs main calculations on the first current element that comes to the input stream. The resulting value is transmitted to the output stream by the operator –>. At the same time, a recursive call to the function `GetStreamResult` is done. The reference to the input stream without the first argument is passed to the recursive function call. The block `block` is used to localize a group of statements, where only one statement returns a result by calling the function `break`. The data come to the block through the identifiers defined outside of it.

When passing the results of computations to a new stream, the order of coming elements can change as compared to their order in the original stream, which leads to nondeterminism of computations and an incorrect result. The example shows that it is needed to extend the computation model with constructs that preserve the order of the data elements, but at the same time support asynchronous interactions.

### *2.2. Ordered Data Element Structures Maintaining the Order*

To preserve the order of data being processed, it is necessary to use container types that provide asynchronous formation of individual elements. In the DDFPCM, such an entity is a parallel list. However, it only supports performing massive operations on its elements and does not allow the list to be processed as a single argument. In the STMDDFPC some extensions are introduced that provide a support for the required functionality. Instead of a parallel list, a swarm is used, which, along with massive operations, like a stream, can be used as a single argument (Fig. 2).

The availability of information about types at compile time is the specificity of the proposed STMDDFPC and Smile, that is, the statically typed language of data driven functional parallel programming developed on the basis of the proposed model. This leads to a change in the algebra of equivalent transformations and the semantics of many basic operations aimed at the generation of code for the target architectures rather than at the interpretation of the source program. In particular, the direct nesting of swarms is prohibited, which makes it easier to parse the arguments of the interpretation operator at compile time and allows determining whether the function is massive over all elements of the swarm or it is a function over the entire swarm. A number of swarm transformations for using in massive operations can be already performed at the compile time. The swarm is denoted by a list of items enclosed in square brackets.

The swarm is passed to and returned from the functions by references, same as streams. The swarm reference syntax in Smile is as follows:

```
swarm_reference = element_type_name "[*]".
```

Using references makes it possible to write the following version of the function for the simultaneous ordered calculation of the sine for all swarm elements:

```
sinSwarm << funcdef X@float[*] -> float[*] {
   X::sin:return
}
```

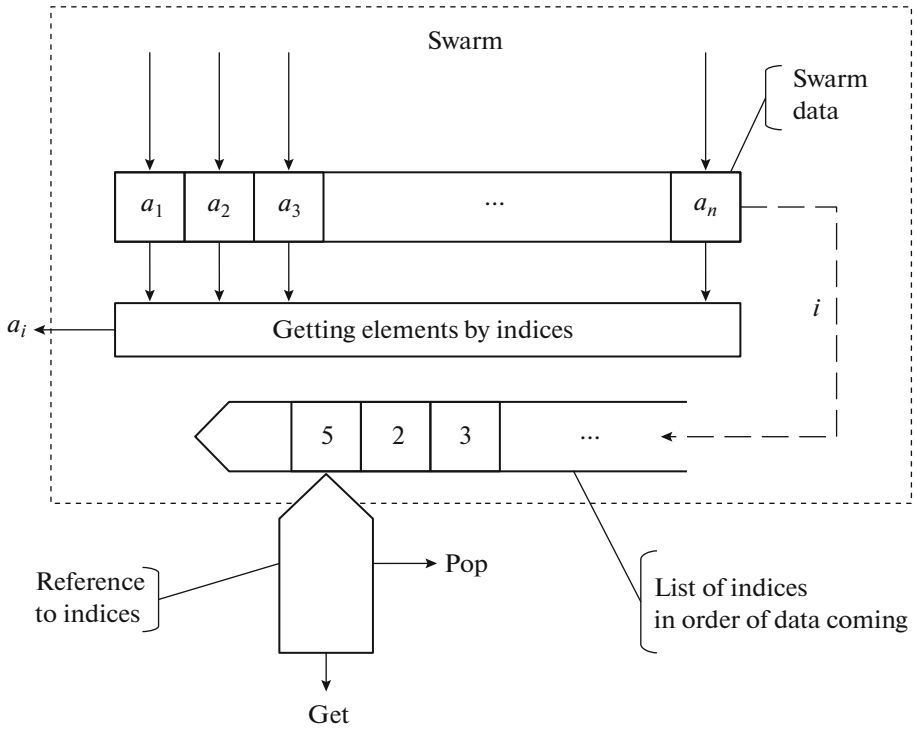


**Fig. 2.** General scheme of a swarm and its reference.

In this situation, the direct use of swarm functions allows getting rid of additional transformations and data synchronization both inside the functions and during their usage:

```
[0.10, 2.1, 0.33, 1.43]:sinSwarm => [0.0998, 0.8632, 0.324, 0.9901]
```

By analyzing the type of the argument of the sinSwarm function, the compiler can easily recognize that it is applied to the whole swarm, but not to each element.

**2.2.1. Using swarms for ordered asynchronous data flow processing.** Unlike streams, even a partially formed swarm has a predetermined size. It can be found at any moment using the size function, the signature of which is as follows:

```
size << func any[*] -> int
```

For example:

```
[10, 21, 33, 43]:size => 4
```

The numbering of the elements of a swarm, like a vector, starts from one. Swarm elements are formed asynchronously. At the appearing of each element the signal is emitted to the associated operator of interpretation, informing about a new available value at a certain index. These indices can be sorted in order of their coming, which, therefore, allows selecting individual elements by these indices sequentially, that is, using an iterator that traverses the elements of the swarm as they appear. As opposed to traversing stream elements, in which access to the elements being created is got directly, getting the index value plays a key role in the swarm. To get index value, the function `get` is used, which has the following signature for the swarm:

```
get << func any[*] -> int
```

It returns the index of the element, that was the first to have come to the swarm.

To move to the next index, the function `pop` is used. It returns a reference to the same swarm with the first index removed:

```
pop << func any[*] -> any[*]
```

Thus, one can iterate over all the elements of the swarm in the order of their coming. In the case when all elements of the swarm are iterated by the reference (in the order of their coming), the function `get` returns zero, which, indicates the completion of the traversal.

In addition, a swarm, like a stream, can be used for sequential processing of asynchronously coming data maintaining the order of the elements in the output. This allows us to rewrite the function for finding `1 - sin(x[i])*sin(x[i])` of the swarm in such a way that it provides the correct order of results in the output:

```
Dif1Sin2Swarm << func X@float[*]->float[*] {
    L << X:size;
    result@float[L];
    (X, result):GetSwarmResult >> ok;
    result:ok:return
}
```

To accumulate data, the function uses the additional swarm storage `result`, which is filled using the principle of a single assignment. That is, a generated code allows writing the data at the same index no more than once. If this rule is violated, the program execution is interrupted. The storage is passed to the function `GetSwarmResult` by reference, in which the storage is filled; then, the obtained value is returned by the function `Dif1Sin2Swarm`. The main calculations are carried out in the function `GetSwarmResult`:

```
GetSwarmResult << func (X@float[*],Y@float[*])->signal {
    i << X:get; // Getting the index of an element from X
    if << (i,0):!=; // Checking for the presence of elements
    if^(
        {block {
          s << X:i:sin; Sin2 << (s,s):*; // Sine squared
          // Calculation of the current value and its transfer to
          // the output swarm by the calculated index
          (1,Sin2):- -> Y[i];
          // Current index is removed from the swarm reference
          // and processing of the next element starts
          (X:{i:signal}:pop, Y):GetSwarmResult:break}
        }, // Putting the result into the second swarm
        // If the index value is equal to zero, then
        // the computation is complete and the signal object is generated
        !
        ):return
}
```

Initially, this function calculates the index of the first element coming into the swarm X. If the value is not equal to zero, then the next index is obtained, which is used to select the *i*th element from the swarm. Then, the difference between one and the sine squared is calculated for this element. The resulting value is put by the reference Y to the *i*th place. The calculations are recursively repeated until the storage `result` is completely filled, which is passed to this function via the reference Y.

**2.2.2. Direct access to swarm elements.** Along with processing swarm elements in the order of their coming, the direct access to elements by index is also possible. If the element has not been placed in the stream yet, then the function waits for its coming. While waiting, a selection of other elements can be initiated using recursive calls in parallel. The disadvantage of this approach is the possibility of many parallel branches waiting for data to come. However, when processing data from multiple swarms, this approach makes computations synchronization easier. The signature of the function used to access an element by index is as follows:

```
base_function<integer> << func any[*] -> any
```

In this case, an integer number in the range from 1 to the size of the swarm is used as a function. If the number is not in this range, then an interruption occurs in the program.

As an example, consider the elementwise product of data passing to the swarm. The function `ScalMultSignal` performs calculations by taking two swarms as arguments by references X and Y. In addition, it receives a reference R to the swarm accumulating the results, as well as a number of elements in the swarm. The latter is used as an index for selecting elements.

```
// The function computing the elementwise product of the swarms
ScalMultSignal << func (X@float[*], Y@float[*], R@float[*], L@int)->signal {
    if << (L,0):!=;
    if^(
        {block{
            (X:L, Y:L):* -> R[L];
            (X, Y, R, L:--):ScalMultSignal:break
        }},
        // Completion of the elements extraction
        !
    ):return
}
```

The multiplication of elements with the same indices is carried out until the transferred value of the index is zeroed by the function "--", which decrements the current value. The recursive call is made immediately after the release of the delayed block, regardless of whether or not the multiplication operation is performed.

Finally, the following function provides an interface for interacting with other functions:

```
// A function used to multiply swarms
// Swarms are assumed to be of the same size
ScalMult << func (X@float[*], Y@float[*]) -> float[*] {
    L << X:size;
    result@float[L]; // Results storage
    ok << (X, Y, R, L):ScalMultSignal;
    result:ok:return
}
```

**2.2.3. Pipelining asynchronous data flow computings.** Passing streams and swarms between functions allows organizing a combination of computations in interconnected functions. As an example, consider the dot product function which uses the element-wise product function of two vectors `ScalMult` and the function `sum` that finds the sum of the elements of the stream:

```
DotProd << func (X@float[*], Y@float[*]) -> float {
    (X, Y):ScalMult:stream:sum:return
}
```

This function takes two swarms and finds the elementwise product. As the results of the multiplication of individual pairs of elements are formed at the output of the function `ScalMult`, they are passed to the stream associated with the input of the function sum. Pipelining in this case is generated automatically depending on the rate of data coming and the rate of operations execution in the function `DotProd`.

## CONCLUSIONS

In the article we consider the method supporting a new approach to the development of parallel programs, which allows describing the parallelism using asynchronous sequentially generated data flows with the control on data readiness. The characteristics of parallelism depend on the rate of data coming and processing. Collaborative use of functions, designed with the proposed computing model, provides support for pipelined computations. It is shown that the proposed methods for data driven functional parallel computing can be implemented in a statically typed language of data driven functional parallel programming.

## CONFLICT OF INTEREST

The authors declare that they have no conflicts on interest.

## REFERENCES

1. Legalov, A.I., The usage of asynchronous lists within the dataflow model of computations, in *The Third Siberian School Seminar on Parallel Computations*, 2006, pp. 113–120.

2. Hoar, C.A.R., *Communicating Sequential Processes,* Commun. ACM, 1978.
3. Diaz, M., *Petri Nets: Fundamental Models, Verification and Applications,* ISTE Ltd., 2009.
4. Legalov, A.I., About computation control in parallel system and programming languages, *Nauchn. Vestn. NGTU*, 2004, vol. 18, no. 3, pp. 63–72.
5. Legalov, A.I., Vasiliev, V.S., and Matkovsky, I.V., Changing computing management strategies for architecture-independent parallel programming, *Proceedings of the XIX All-Russian Scientific Conference Scientific Service on the Internet,* 2017, pp. 341–350.
6. Redkin, A.V. and Legalov, A.I., Event based control of computations for functional dataflow programming, *Sci. Bull. Novosib. State Tech. Univ.*, 2008, vol. 32, no. 3, pp. 111–120.
7. Legalov, A.I., Redkin, A.V., and Matkovsky, I.V., Data driven functional parallel programming with data coming asynchronously, in *Parallel Computing Technologiws (PCT'*2009*)*, 2009, pp. 573–578.
8. Legalov, A.I., Legalov, I.A., and Matkovsky, I.V., Specifics of semantics of a statically typed language of functional and dataflow parallel programming, in *Scientific Conference Scientific Service on the Internet,* 2019, pp. 274–284.
9. Legalov, A.I., Functional language for architecture-independent programming, *Comput. Technol.*, 2005, vol. 10, no. 1, pp. 71–89.