Asynchronous-streamed model for describing dynamically changing parallelism

A I Legalov[1] **and D S Romanova**[1]

[1] Siberian Federal University, 79, Svobodny pr., 660041, Krasnoyarsk, Russia

E-Mail: legalov@mail.ru, daryaooo@mail.ru

Abstract. The article presents the concept of an asynchronous-streamed model, which is an extension of the functional data-flow model of parallel computing. We suppose that the model will be used to describe dynamically changing parallelism. This model is based on the concept of asynchronous lists, which allow considering their parallelism as an independent way of describing the parallelism of the program. The specifics of asynchronous lists are represented. Using the examples, we show how to use this model to get temporary estimates of parallelism at different conditions of solving the problem. These estimates depends on correlation between data speed getting and computation time those data in parallel devices. Also, we demonstrate how two connected asynchronous-streamed functions can form pipeline automatically. As a result, we show out that on this model it is possible to obtain temporary estimates of the calculations performed and estimations which demonstrate the levels of parallelism.

1. Introduction

Parallel programming offers different approaches to create parallel programs. Most of these approaches depend on concrete parallel computer architecture. Parallel computing systems use different methods of organizing parallel processes. For example, they oriented to execute processes of various sizes, have different methods of memory access and methods of transmitting information between memory and processors. They use different ways of managing computer resources too. The most popular are imperative programming paradigms.

To estimate parallel computing, the models that describe various aspects of parallel programs are used. These models make it possible to evaluate parallel programs by different parameters. For example, we can estimate such parameters as the parallelism level of program or the relation's of execution time between operations, or the efficiency using of computing resources. There are models, which focused on the evaluation of parallel computing. These models do not allow writing parallel programs directly, but they provide various estimations and variants of analysis. In addition, they define various paradigmatic approaches that lead to the development of parallel programming languages.

Some abstract models don't take into account real computing resources, assuming that they are unlimited. In this case, it's possible to analyze the maximal parallelism and evaluate a level of parallelism by various parameters. Besides, these models provide a conceptual basis for developing of architecture-independent parallel programming languages. Also, they can be used for creating methods of transforming architecture-independent parallel programs to programs for real parallel computers. These models can descript parallel processes using elementary operations. Each operation can be evaluated in its own computing resource. It allows avoiding of resource conflicts. And it is often unimportant which computation management strategy was used [1].

The following models can be distinguished:

- Model of communicating sequential processes [2];
- Q-determinant [7];
- Functional data-flow parallel computing model (FDFPCM) [1].

These models are used to solve different problems.

2. Hoare's model of communicating sequential processes

The model describes the communication of processes using a mechanism for transmitting asynchronous messages [2]. The processes, in fact, are computing resources and may be reusable during calculations. This conception is being formed as is to a greater extent a model of interaction between limited resources. Based on this, various conflicts can arise between processes.

The main feature of this model is the use of an event as an instantaneous action that fixes the beginning or end of a process. The event is defined as some signal, which is sent from predecessor to successor and carry additional identifying information. Therefore, events' transmission defines parallel system as a set of interacted processes which communicate using signals. The model describes the synchronizing primitives, which allow resolving of resource conflicts as one of results. The solution of various problems of communication of processes is considered in [2] including well-known problems of their synchronization. This model formed the basis of real software and hardware solutions, for example, the Occam programming language [2, 3], transputer systems [4], and Go-programs of the Go programming language [5].

A key feature of the model is that the moments of occurrence of events in time are sequential. That is, events never occur simultaneously and do not overlap. Instead of simultaneity, an interpretation from a nondeterministic appearance is used when their order is not defined and can be arbitrary. A similar assumption is also used in Petri nets [6], the classical apparatus of which is aimed at studying the behavior of parallel systems.

3. Q-determinant

In the contrast to Hoare's model, the concept of Q-determinant [7-9] is not based on event communicating of processes. Parallelization and parallel execution of algorithms, taking into account the independence of operations, formed the basis of this model. Q-determinant can be used in solving the problem of describing algorithms with maximal possible parallelism. This approach allows to study the parallelization resource of the algorithm, in particular, to reflect the existing parallelism of the algorithm and to demonstrate the possible way of its execution. It's possible to obtain Q-effective implementations of the algorithm for real parallel computing systems [8]. So, this is an opportunity not only to describe algorithms efficiently, but also to increase this efficiency of the implementation of methods for solving algorithmic problems. The attempt to create a mathematical apparatus for solving the following problems:

- Determination of the most efficient implementation of the algorithm based on this concept and assessment of the complexity of this implementation;
- Maximal parallelization of algorithms [8];
- Determination and analysis of the parallelization resource of some algorithms (scalar product of vectors, matrix multiplication, Gauss-Jordan and Jacobi methods of solving of linear equation systems) [9];
- Method of constructing a Q-determinant based on its flowchart was developed;
- Method for calculating and comparing the characteristics of parallel complexity of a Q-efficient implementation of the algorithm was created [9].

Based on this, the main purpose of the model is to evaluate parallelism. So, to make it possible, the Software system QStudio was created [9]. It helps to evaluate a particular implementation written using the Q-determinant model.

At the same time, the lack of interaction between processes does not clearly allow determining how parallelism will be described further in solving certain problems.

4. Dataflow functional model of parallel computing

The DataFlow Functional Model of Parallel Computing (DFFMPC) is oriented on an architecture-independent representation of parallel computing using unlimited resources. The model was used in the Pifagor language development [11]. This model directly relates to its transformation into a Control-Flow Graph (CFG), in which control signals are, in fact, equivalent to events in the Hoar's model. And these signals are executed instantly, while signaling the start of execution of the data readiness control. Based on this model, the tools were constructed that provide support for compiling, assembling, debugging, and executing functional dataflow parallel programs. Researchers are being carried out related to the verification, optimization and transformation of programs into other programming languages and systems on a chip [10, 11]. The obtained intermediate representation in the form of a Reverse Data Flow Graph (RDFG) and CFG make it possible to evaluate time of program execution.

The model contents asynchronous lists that allow considering parallelism not as a static phenomenon, but as an independent way of parallelism describing [11]. This parallelism may be changed dynamically. It depends on the time estimates associated with performing of various operations, transferring data between memory channels, and the speed at which data arrives to memory. In this regard, a subset of the DFFMPC can be distinguished, which we call as the asynchronous-streamed model.

5. Asynchronous-streamed model

At the same time, the lack of interaction between processes does not allow to clearly determine, how parallelism will be described further in solving certain problems. This model is based on the data presentation in the form of asynchronous lists, which can also be defined as streams. Stream is ready when there is even only one data element of it [12]. Each of the values in this stream generates a signal of readiness that can be processed by the operation of an interpretation. The data generated in the stream have the appropriate structure. The mechanism of this model is universal and may be used in different algorithms. In this case, sequential events can be described recursively by using left and right recursion.

5.1. Definition of asynchronous-streamed model

Let the asynchronous list be denoted as A_N, where N is the number of elements generated in it. Then its structure can be described by the following recursive expression:

$$\text{If } N = 0, \text{ then } A_0 = (.),$$
$$\text{If } N>0, \text{ then } A_N = (d, A_{N-1}).$$

Where d is the head element of the generated list A_{N-1} is the asynchronous list of the remaining N-1 elements, A_0 is an empty asynchronous list and (.) is an empty data list [11].

Thus, for each of the interpretation operations, a list of data consisting of formed element and asynchronous list in which other elements accumulate. This "tail" may not yet be formed by the time the first element appears.

If a stream issued all data, then it is considered as completed. If the data is absent in the stream, and then an empty data list is as result. The same empty data list terminates any stream. Denote the stream by

$$A_N = \text{stream}(d_1, d_2, \dots d_N)$$

The result is a gradually formed data list, which consists of pairs nested in each other:

$$A_N = \text{stream } (d_1, d_2, d_3 \dots d_N) \Rightarrow (d1, \text{stream } (, d_2, d_3 \dots d_N)) \Rightarrow$$

$(d_1, (d_2, \text{stream } (d_3, \dots d_N))) \Rightarrow (d_1, (d_2, (d_3, \dots (d_{N-1}, \text{stream } (d_N))) \Rightarrow$
$(d_1, (d_2, d_3, \dots (d_N, \text{stream } (.))) \Rightarrow (d_1, (d_2,d_3 \dots (d_N, (.))).$
$A_N = \text{stream } (d_1, d_2,d_3 \dots d_N) \Rightarrow (d1, \text{stream } (, d_2,d_3 \dots d_N)) \Rightarrow$
$(d_1, (d_2, \text{stream } (d_3, \dots d_N))) \Rightarrow (d_1, (d_2, (d_3, \dots (d_{N-1}, \text{stream } (d_N))) \Rightarrow$
$(d_1, (d_2, d_3, \dots (d_N, \text{stream } (.))) \Rightarrow (d_1, (d_2,d_3 \dots (d_N, (.))).$

The creation of the next pair is possible only when the previous pair will be formed. That means, there can't be direct access to an arbitrary element of the asynchronous stream. If all data of a stream were generated in advance, then their order in the stream is determined by the placement in the original data list. For example a one-dimensional array X = (44, 55,12,42) may be transformed to asynchronous stream from the original data the following way:

stream (44,55,12,42) => (44, stream (55,12,42)) => (44, (55, stream (12,42))) =>
=> (44, (55, (12, stream (42)) => (44, (55, (12, (42, (.))))).

Thus, next data element can be used for processing in each of the operations of interpretation. And we can select the remaining elements as a new asynchronous list, for processing by next interpretation operation, if it possible. At the same time the dataflow control principle is preserved. And the processing of the tail of the list does not depend on the moment when data will be arrived.

We can see the differences between regular and asynchronous approaches using a simple example of addition vector elements. The traditional approach to addition vector elements is based on cascading convolution. The general scheme of this approach is presented in Figure 1.

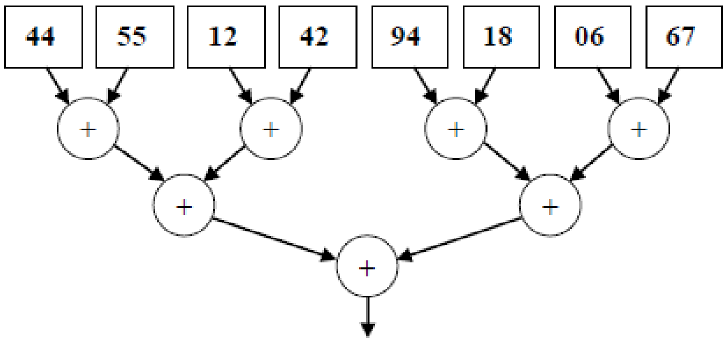

Figure 1. The cascading scheme of addition vector elements

But often the vector may not be known at the beginning of the calculation, and the data may arrive in random order. The time interval between the generated events in asynchronous models is not significant. That is, it can be large or close to zero. In this case, the most important role is played by the ratio of the intervals between the moments of data generation and the duration of the computational operations. These times can be comparable or correlated as very large and very small quantities. In the latter case, very small quantities can be neglected, equating them to zero.

Based on this, the following scheme for calculating the sum of the elements of a one-dimensional array based on data availability can be presented. The elements of the array, asynchronously formed during the calculations, form a queue, into which they are entered in the order of appearance. If there are two or more elements in this queue, the pairs formed are summed. Then they are sent to the same queue. The only element remaining during the summation is the final result. The scheme of such organization of calculations is presented in Figure 2.

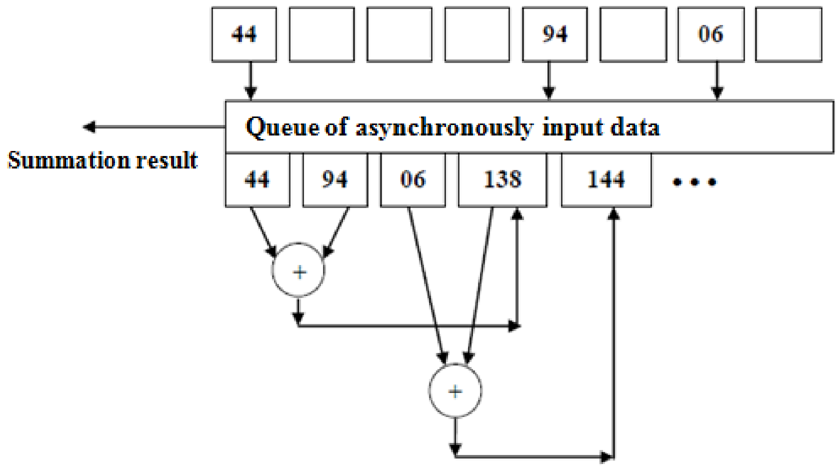

Figure 2. Summing asynchronously input data

5.2. Using asynchronous lists in the programming language Pifagor

The above extensions to DFFMPC can implement in the programming language Pifagor. This will allow writing parallel programs in a slightly different style, which takes into account asynchronous receipt data. In particular, the addition of elements of a one-dimensional array, corresponding to the scheme, presented in Figure 1, can be implemented as follows:

```
// Function which returning the summation of asynchronous list's elements
A_VecSum<< funcdef A
       // format of the argument: A=stream(x1, x2, … , xn)
       // где x1, x2, …,xn are numbers
{
    x1<< A:1; // input element of data
    tail_1<< A:2; // tail of asynchronous list
    // Checking if the list is empty
    [(tail_1:[IsEmptyDataList, IsNotEmptyDataList]:?]^
    (
       x1, // In the list, there is only one element that determines the
       amount
       {
              // Determining the second argument followed by summation
              block {
              x2<< tail_1:1; // the second argument
              s<< (x1,x2):+; // sum of two received elements
              tail_2<< tail_1:2; // tail for the tail
              // Recursive processing of the remaining elements
              [(tail_2:[IsEmptyDataList, IsNotEmptyDataList]:?]^
               (
                  s,
                  { stream( tail_2:[], s):A_VecSum }
                ):. >>break
              }
     }
   ):. >>return; }
```

It should be noted that when this function arrives at the input of a regular data list instead of an asynchronous list, it will be correctly processed. This is due to the equivalent interpretation of the operations of taking the first element and highlighting the sublist without the first element for both lists.

5.3. Interpretation of time relationships of an asynchronous streaming program

Using asynchronous lists allows developing a program that, depending on the time relationships between its operations, can be interpreted as a set of alternative algorithms that describe the same task. For example, if the time interval Δt_{dat} between the generation of data inside the asynchronous list is longer than the execution time of the addition operation Δt_{add}, then the summation will be performed sequentially. This situation is illustrated in Figure 3.

It should be noted that in this case, we are not talking about specific temporal characteristics, but about the relation of times at the level of an abstract computational model.

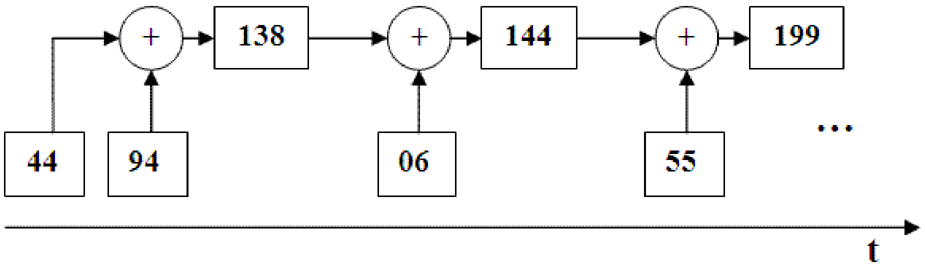

Figure 3. Sequential summation with $\Delta t_{dat} >= \Delta t_{add}$

In the event that the time interval for the receipt of data becomes less than the time of their addition, parallelism will begin to appear in the performance of these operations. The number of parallel threads will be a multiple of the ratio of time Δt_{add} to Δt_{dat}. For example, with

$\Delta t_{add} / \Delta t_{dat} = 2$, two addition operations flows will be dynamically generated (Figure 4).

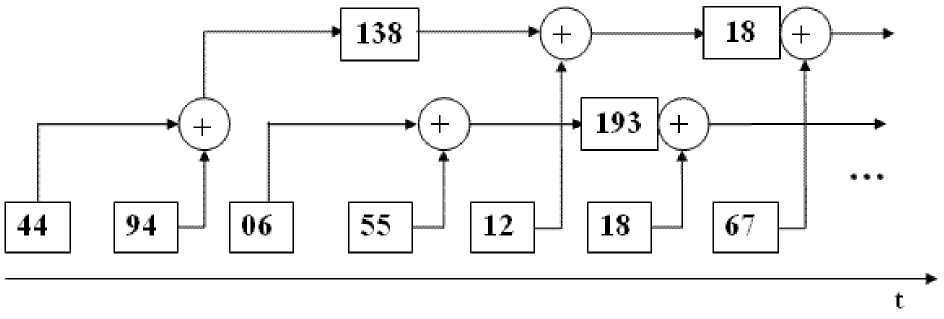

Figure 4. Formation of two parallel flows with $\Delta t_{add} = 2 * \Delta t_{dat}$

If the rate of data receipt is much higher than the time of the addition operation, then we can assume that they have time to arrive before the start of the summation, which will begin almost simultaneously for all pairs of numbers [11]. Upon completion of the summation, the results also "instantly" enter the asynchronous list, thereby ensuring simultaneous summation on the second layer. In this case, the general scheme of calculations will correspond to the tree convolution shown in Figure 1.

In this work, we will also focus on time relationships when solving a problem using asynchronous lists. As an example, consider the pairwise multiplication of two vectors.

Two vectors of the same length A_i and B_i arrive at the input to the processor D, and their arrival rates are arbitrary and are determined as Δt_{da} and Δt_{cb}, respectively. As a result, a vector is formed at the output, while data race and state changes are possible.

The solution to this problem consists of two steps:

- Decomposition;
- Convolution [12].

Let us consider in more detail the decomposition step, during which subproblems of the form A_k * B_k (k = 1... n) are formed.

When it is implemented in asynchronous mode, the data arrives with a certain intensity Δt_d and is processed at a speed determined by the interval Δt_c. If the data arrives faster than the operation time, then new resources can be allocated for the input data. Otherwise, one resource is enough to solve the problem. If the data arrives simultaneously, then the number of operations is determined by the number of pairs received. In general, it can be noted that the amount of resources allocated at a certain step of solving the problem is determined by time relationships.

In this case, the number of resources R that must be allocated to the system to solve this problem is determined by the formula:

$$\frac{\Delta t_d}{\Delta t_c} = k,$$

where k is the coefficient of time relations.

This formula is valid for uniform calculations.

Depending on the values that this coefficient will take, the following cases are possible:

1. If $\Delta t_d \leq \Delta t_c$, then 0 <k <1, then only one resource is needed to solve this stage of the problem (R = 1).
2. For example, if 2 < k < 3, then k will determine the number of simultaneously involved resources at this stage of solving the problem = R, in this situation the number of allocated resources will be two.
3. For the largest possible number of allocated resources, the coefficient k must be a large number, i.e. $\frac{max\Delta t_d}{min\Delta t_c}$. Therefore, it is possible to determine by this coefficient how many data pairs can be processed simultaneously. This number of pairs (n_s) will be determined depending on the maximum number of R_{Smax} resources allocated at this step, i.e. $n_s = R_{Smax}$.
4. To achieve maximum parallelism in solving the problem, it is necessary that the number of multiplication operations at this step be approximately equal to the number of allocated resources R, i.e.

$$n \approx R,$$

where n is the number of operations.

In the general case, taking into account the time characteristics, the following conditions for the maximum parallelism of the problem can be written:

a) $\frac{max\Delta t_d}{min\Delta t_c}$

b) $\Delta t_d \geq n$, where n is the number of input data pairs of the form (A_m, B_m), m=1, …, n

The scheme of asynchronous calculations at the step of decomposition of the solution to this problem, taking into account the time characteristics, is shown in Figure 5. In this case, we are talking about computing nodes (operating devices) that provide processing of input data.

Software implementation of the decomposition step in the Pifagor language is shown here:

```
// call the main decomposition function
// Data input format: (list), (list)
decCaller << funcdef X
 {
       return << (X: 1, X: 2,1,1, (.)): dec;
 }
 // function for finding subtasks at the decomposition step
 dec << funcdef X
  {
       A << X: 1; // input vector A
       B << X: 2; // input vector B
```

```
        i << X: 3; // counter
        Len << A: |; // the length of the vector A (provided that the
        vectors A and B are the same length)
        c << X: 4;
        list << X: 5;
        [((i, Len): [<=,>]):?] ^
         (
               {(A, B, (i, 1): +, (A: i, B: i): * >> mult1, (list, mult1):
        concat): dec},
               {list}
          ) :. >> return;
  }
 // helper function for generating an asynchronous list
 concat << funcdef X
  {
        list << X: 1; // data item
        tail << X: 2; // tail of the list
    return << stream (list, tail); // returning asynchronous list for
    future calculations
  }
```

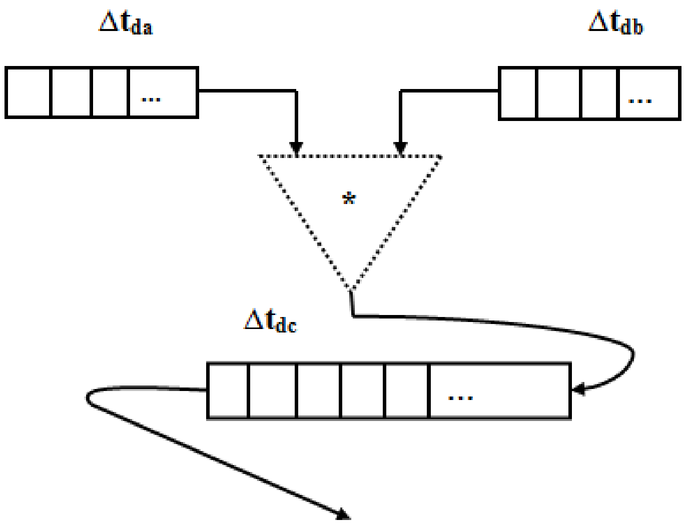

Figure 5. Asynchronous computing diagram at the decomposition stage of solving the problem of pairwise multiplication of two vectors

With a consistent description, parallelism can be described at various levels. In addition to of temporary estimates of parallelism, one can obtain theoretical estimates of maximum parallelism associated with tiering. These estimates can be obtained by assuming that all data comes simultaneously and instantly. That is, assume that $[\Delta t_d] = 0$. In this case, any length of data processing time will be much longer than the time of their arrival. Assuming $[\Delta t_d] = 1$, we can calculate the number of tiers of the parallel algorithm in the same way as in the model based on the Q-determinant [8].

When all the data is processed at such a rate that at the output of the processor it appears simultaneously, then a tiered assessment of maximum parallelism is taken place, where 1 is the type of tier.

If, after processing at the first step, multiple simultaneous processing takes place, and the number of such operations is equal to n, then a tiered estimate can be expressed by the equation:

$T=\log(\Delta t_{c1})$, where Δt_{c1} is the processing time at the first step and $\Delta t_{c1}=\Delta t_{c2}=\ldots=\Delta t_{cn}$.

The study of this approach can be considered on more complex algorithms, which show that in this case, pipelined and other estimates are possible. As an example, we can consider the algorithm of vector multiplication, which gives output 1.

In this algorithm, pairwise multiplication is first performed (the scheme is shown in Figure 5), and then further summed up using the convolution operation (scheme in Figure 6). The order of summation is arbitrary.

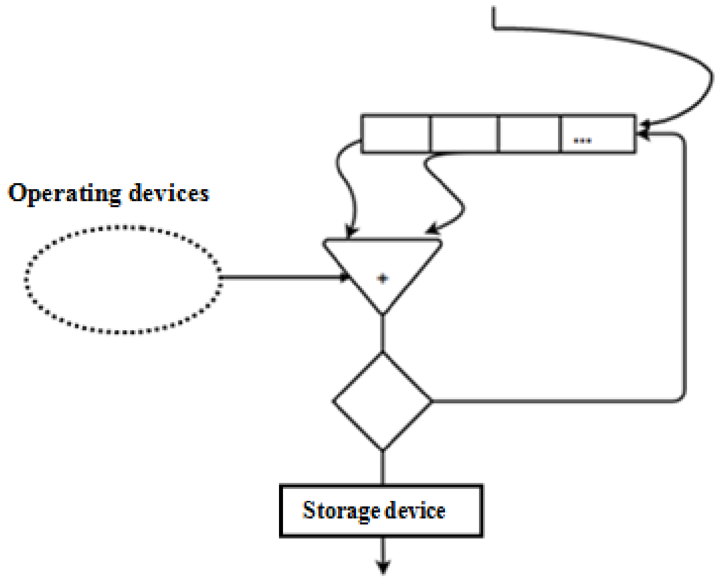

Figure 6. General scheme for solving the problem of the convolultion step of pairwise multiplication of vectors

Based on this, it is clear that estimates are related to the parallelism of these two problems. For time-limited estimates in the addition algorithm, parallelism will change as well as in the multiplication problem, from maximum to sequential. This allows obtaining different algorithmic estimates on the same description and evaluating the algorithm from the point of view of the same description.

This task shows that in this situation, pipelining of calculations is possible. Here the input data can already undergo a summation operation during the continuation of the whole operation, and not one after another, in contrast to the case with abstract estimates that lead to simultaneity. But in real calculations it is not so.

It should be noted that all of these studies can also be applied to solving the problem of matrix multiplication, since it reduces to vector multiplication. In this case, a full time or tier assessment can be used.

6. Conclusion

The use of asynchronous lists allows implementing a new class of algorithms, the parallelism of which depends on the time relationships between the basic and preparatory operations. This allows talking about a specific approach to the construction of parallel functional-stream algorithms, with the help of which we can develop and research new methods for creating portable parallel programs. Unlike equivalent transformations of generalized functions proposed in [1] for adapting a parallel algorithm to a specific computing system, the use of asynchronous lists makes it possible to use the same algorithm on which different time relationships between the operations performed are superimposed.

The proposed mechanism for describing asynchronous computing provides the programmer with additional features. Its instrumental support at the language level ensures the use of new methods for developing functional-stream parallel programs. They are characterized by a dynamic change in parallelism depending on the time relationships between the operations performed. This increases the flexibility of the developed algorithms, and allows using the same algorithm for dynamically setting different levels of parallelism. The programs formed at the same time allow us to describe the maximum parallelism of the problem being solved and are independent of the architectures of the computing systems used. However, it should be noted that the use of asynchronous lists does not affect the organization of a virtual computer used for parallel interpretation of functional programs.

As a result of the research, the extension of the functional-stream computation model was proposed. It provides a description of parallelism formed in the form of a recursion operation on asynchronous lists, which can be deployed into dynamically generated parallelism. And at the same time, this parallelism can be performed in parallel depending on time relationships. Using this model, both temporary estimates and estimates connecting the tiers of parallelism using abstract relations in the form of abstract intervals can be obtained.

The reported study was funded by RFBR according to the research project no. 17-07-00288.

References

[1] Legalov A I 2001 *Distributed and cluster computing. Selected materials of the School-Workshop* Management strategies in computing systems and programming languages (Krasnoyarsk: Int. of Comp. Modeling SB RAS) p 94-108

[2] Hoare C A R 1985 *Communicating Sequential Processes* (UK: Prentice Hall International) p 260

[3] Carling A 1988 *Parallel Processing,Transputer and Occam* p 162 (Sigma Press)

[4] Patt Y N Patel S J 2003 *Introduction to computing systems, 2nd Ed.* (New York: McGraw-Hill) p 575

[5] Legalov A I 2005 *Computational Technologies* The functional programming language for creating architecture independent parallel programs vol **10** no 1 (Novosibirsk: Institute of Computational Technologies SBHAS) pp 71–89 (In Russian)

[6] Diaz M 2009 *Petri Nets: Fundamental models, Verification and Applications* (UK: ISTE Ltd) p 585

[7] Aleeva V N 2019 *Materials of 71st Sc. Conf. Section of technical sciences* Main ideas of technology of Q-determinant (Chelyabinsk: UYrGU) p 334–42

[8] Aleeva V 2018 *Supercomputing RuSCDays Communications in Comp. and Inf. Sc.* Designing a Parallel Programs on the Base of the Conception of Q-Determinant vol **965** Voevodin V, Sobolev S (eds) pp 565–77

[9] Aleeva V, Bogatyreva E, Skleznev A, Sokolov M, Shuppa A 2019 *Supercomputing RuSCDays Communications in Comp. and Inf. Sc.* Software Q-system for the Research of the Resource of Numerical Algorithms Parallelism vol **1129** Voevodin V, Sobolev S (eds) pp. 641–52

[10] Udalova U V 2011 *Jornal of SFU* Methods of debugging and verification of data-flow parallel programs vol **4** no 2 pp 213–24

[11] Kropacheva M, Legalov A 2013 *Parallel Computing Technologies, 12th Int. Conf. PACT September-October, St. Petersburg* Formal Verification of Programs in the Pifagor Language (Lecture Notes in Computer Science 7979, Springer) pp 80 – 89

[12] Ananth G, Anshul G, Karypis G and Kumar V *2003 Introduction to Parallel Computing* (USA: Addison Wesley) p 856