Methods and Tools of Parallel Programming

Victor Kasyanov and Elena Kasyanova

Institute of Informatics Systems, 630090 Novosibirsk, Russia,
Novosibirsk State University, 630090 Novosibirsk, Russia
kvn@iis.nsk.su

Abstract. Using traditional methods, it is very difficult to develop high quality, portable software for parallel computers. In particular, parallel software cannot be developed on low cost, sequential computers and then moved to high performance parallel computers without extensive rewriting and debugging. In this paper, the CSS system being under development at the Institute of Informatics Systems is considered. The CSS is aimed to be an interactive visual environment for supporting of cloud parallel programming. The input language of the CSS system is a functional language Cloud Sisal that exposes implicit parallelism through data dependence and guarantees determinate result. The CSS system provides means to write and debug functional programs regardless target architectures on low-cost devices as well as to translate them into optimized parallel programs, appropriate to the target execution platforms, and then execute on high performance parallel computers in clouds without extensive rewriting and debugging.

Keywords: cloud computing, computer science education, functional programming, hierarchical graph representation, parallel programming.

1 Introduction

Using traditional methods, it is very difficult to develop high-quality, portable software for parallel computers. In particular, parallel software for supporting of enterprise information systems cannot be developed on low-cost, sequential computers and then moved to high-performance parallel computers without extensive rewriting and debugging. Functional programming [1] is a programming paradigm, which is entirely different from the conventional model: a functional program can be recursively defined as a composition of functions where each function can itself be another composition of functions or a primitive operator (such as arithmetic operators, etc.). The programmer need not be concerned with explicit specification of parallel processes since independent functions are activated by the predecessor functions and the data dependencies of the program. This also means that control can be distributed. Further, no central memory system is inherent to the model since data is not “written” in by any instruction but is “passed from” one function to the next. However, scientific world is conservative and the FORTRAN programming language is still quite popular in scientific computations for supercomputers.

Functional language Sisal (Streams and Iterations in a Single Assignment Language) is considered as an alternative to FORTRAN language for supercomputers [2],[3]. Compared with imperative languages (like FORTRAN), functional languages, such as Sisal, simplifies programmer's work. He has only to specify a result of calculations and it is a compiler that is responsible for mapping an algorithm to certain calculator architecture. In contrast with other functional languages (like Lisp, ML and Haskel), Sisal supports data types and operators typical for scientific calculations such as loops and arrays. At present, there are implementations of the Sisal 1.2 language [4] for many supercomputers (e. g., SGI, Sequent, Encore Multimax, Cray X-MP, Cray 2, etc). Sisal 90 [5] language definitions increase the language's utility for scientific programming and include language level support for complex values, array and vector operations, higher order functions, rectangular arrays, and an explicit interface to other languages like FORTRAN and C. The Sisal 3.2 language [6],[7] integrates features of Sisal 2.0 [8] and Sisal 90 versions and includes language level support for module design, mixed language programming, and preprocessing.

In this paper, a visual cloud system CSS (Cloud Sisal System) of functional and parallel programming being under development at the Institute of Informatics Systems is considered. The input language of the system is a functional language Cloud Sisal [9] that is a modification of our Sisal 3.2 language which is aimed to increase the language's utility for supporting of cloud scientific computations and cloud parallel programming. The Cloud Sisal language supports also so-called annotated programming and concretizing transformations [10], [11]. The system presented uses intermediate languages of hierarchical graphs and provides means to write and debug functional programs regardless target architectures on low-cost devices as well as to translate them into optimized parallel programs, appropriate to the target execution platforms, and then execute them on high performance parallel computers in clouds without extensive rewriting and debugging [12], [13]. So, the system can open the world of parallel and functional programming to all students and scientists without requiring a large investment in new, top-end computer systems.

2 The CSS System

The advancement of computer technology and the increasing complexity of research problems are creating the need to teach parallel programming in higher education more effectively. Programming massively-parallel machine is a daunting task for any human programmer and parallelization may even be impossible for any compiler. Instead, the functional programming paradigm may prove to be an ideal solution by providing an implicitly parallel interface to the programmer.

The CSS system is based on hierarchical graph representations of functional and parallel programs and is intended to provide a general-purpose user interface for a wide range of parallel processing platforms (See Fig. 1). In our conception, cloud interface gives transparent ability to execute programs on arbitrary environment. JavaScript client does not demand installation; small educational

programs can be executed on client devices (computers or smart phones). V8 server allows the language parser and some optimizations to be used at both client and server sides.

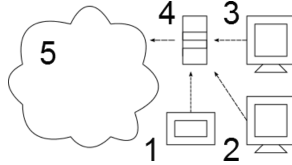

Fig. 1. Cloud service structure: 1, 2 and 3 - clients, 4 - cloud access server, 5 - execution environment.

The CSS system uses a functional language Cloud Sisal and includes five big parts: interface, interpreter, graphic visualization/debugging subsystem, optimizing cross compiler, cluster runtime.

The interpreter is available on web via browser; it translates Cloud-Sisal-program to the first internal representation (IR1) and runs it without making actual low-level code. It is useful because in this case a user can get any debugging information in visual forms of hierarchical graphs [12], [14]. Web interface also contains some usual parts like syntax highlighting, persistent storage for program code, authorization and so on.

The CSS system provides means to write and debug Cloud-Sisal-programs on low-cost devices as well as to translate and execute them in clouds. The CSS system can open the world of parallel and functional programming to all students and scientists without requiring a large investment in new, top-end computer systems.

3 Cloud Sisal Language

The Cloud Sisal language that has been designed as the input language of the CSS system is a modification of our Sisal 3.2 language which is aimed to increase the language's utility for supporting of cloud scientific computations and cloud parallel programming [6],[7], [9],[15].

3.1 Single Assignment

Cloud Sisal differs from other functional languages and we think that this difference make Cloud Sisal more adapted for computational tasks. First of all, it has some usual functional language benefits like single assignment [16]. This approach requires every variable to be defined only once. Someone would say that it is not an advantage because every imperative program can be converted to SSA-form, and of course at low-level programming it has no difference but imagine some function and the global variable in the language where every variable need to be declared (we use C for example):

```
int g=0;
void foo(void) { g=1; }
```

You need to re-declare the global variable when it is modified, but you can't make it inside the function. Inside the compiler this program will be converted quite easy but to write initially singe assignment programs is not the same. You can declare another global variable without setting any value but it can bring more questions to the rest of the code, we can use more complex example to withdraw this but we wouldn't. The idea is that single assignment is something similar to structural programming where "goto" operator is prohibited.

3.2 Streams and Arrays

Cloud Sisal also uses arrays and loops which is not common for a functional language, but it is good for computation: you don't have to worry about the recognition of the tail recursion or the number of iterations or matrix description which is simpler with arrays. You can operate with n-th element of the array in a natural way like in FORTRAN:

```
for i in 1, N repeat
    R := A[i] * B[k]
    returns array of R
end for
```

3.3 Verbose Syntax

And the last benefit is more verbose syntax. It makes program source more readable and as the result - long time development by different people becomes easier. Many functional languages suffers from the lack of the words in the program source, it makes the text hard to understand. The example below is the famous Haskell quicksort:

```
qsort [] = []
qsort (x : xs) =
    qsort[y|y <- xs, y < x] ++ [x] ++ qsort [y|y <- xs, y >= x]
```

This kind of code is hard to maintain. The same algorithm implemented in Cloud Sisal listed below:

```
function qsort (Data:array[real] returns array[real])
 if array_size(Data) > 2 then
  let
     L, Middle, R := for E in Data
     returns
       array of E when E < Data[1]
       array of E when E = Data[1]
       array of E when E > Data[1]
```

```
      end for
   in
      qsort(L) || Middle || qsort(R)
   end let
  else
   Data
  end if
end function
```

3.4 Loops and Reductions

In functional programming every statement is a function returning the value, the Cloud Sisal loops are the same. Reduction is used to determine the returning value of the loop. Keyword "returns" at the end of the loop is followed by the name of the reduction and its parameters. For example, if we need to summarize the elements in the array or the stream we can use the following function with "for all" loop:

```
function sum(A: array[real] returns real)
   for r in A
    returns sum of r
   end for
end function
```

Of course, loop construction can be used without any function declaration. Cloud Sisal is pure functional, it has no side effects and any loop contains the reduction call, also user can implement his own reductions.

The reductions are good because its implementation can depend on target system. When the program is executed in single-threaded environment it can be performed sequentially, but when executed on multiple threads it can be performed in parallel. Similar idea can be found in modern library "Threading Building Blocks" by Intel. This library allows usage of reduction mechanism in C++, but user can also use ordinary loops as well. In Cloud Sisal programs reductions can't be avoided.

In Cloud Sisal we have three kinds of loops: post-conditional, pre-conditional and "for all" (operation is applied to a set). Reductions can be folding or generating (some aggregation function or an array generator). Conditional loops are sequential in general but reduction allows them to be pipelined easier (Fig. 2, Fig. 3). Loops can be divided into parts: initialization, loop body, loop test, loop reduction (ret) and range generator. We think that the part names can briefly describe them, but if you need more information - please check Sisal 3.2 or Cloud Sisal language description [6], [9].

Using reductions matrix multiplication can be implemented meaningfully:

```
function multiply( A,B : array[array[real]];
      M,N,L : integer
```

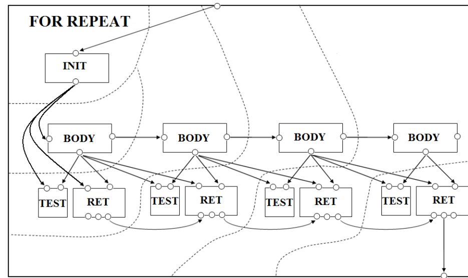

Fig. 2. Post-conditional (for repeat) pipelined structure.

```
      returns array[array[real]] )
  for i in 1, M cross j in 1, L
    returns array of
      for k in 1, N repeat
          R := A[i,k] * B[k,j]
          returns sum of R
      end for
  end for
end function
```

Reduction can be always used in sequential style:

```
function multiply (A:array[array[real]];
   B:array[array[real]];
   N:integer
   returns array[array[real]])
 for i in 1, N cross j in 1, N
   returns array of
    for
      initial s := 0.0; k := 1
      while (k <= N)
    repeat
       s:=old s + A[i,old k]*B[old k,j];
       k := old k + 1
      returns value of s
    end for
 end for
end function
```

But imperative languages doesn't have any reduction mechanism at all.

3.5 Error Handling

Try-catch mechanism is more popular for error handling today but this approach has conflicts with parallel program execution. When the exception occurs all the

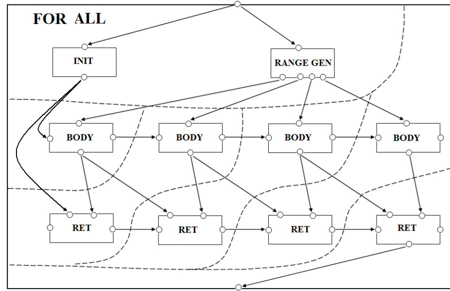

Fig. 3. “For all” pipelined structure.

execution streams must be stopped, pipeline flushed and so on. Also it is harder to keep program determinism in the case of the parallel execution and exception occurs. Check the following Java example:

```
try {for (int i=0; i<N; i++) {a[i]= a[i]/((i+1) % K);}
  } catch (Exception e) {
//display partial results stored in ‘‘a’’
}
```

In this example loop iterations are independent and can be executed in parallel. Sequential execution will always give the same result (for the fixed values of N and K); the result will not depend on the executor properties as far as it remains to be sequential. While there is no dependence between the iterations, programming language semantics remains to be sequential and parallelism exploration can break this semantics or demand additional corrections to keep it. Interpreter or parallelizing compiler needs additional mechanism to differ between the data before and after the exception.

In Cloud Sisal language we have “always finished computations” semantics, which means that execution stream will not stop on any error and return resulting value even if the error occurs (Fig. 4).

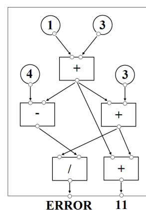

Fig. 4. Error value propagation in “always finished computations” semantics.

3.6 Optimizing Annotations

The Cloud Sisal language supports also so-called annotated programming and concretizing transformations [10] and includes pragma statements in the form of formalized comments (annotations) that start with dollar sign '$' and are predicate constraints on admissible properties of program fragments or states of computations. In addition to restricted set of program executions and restricted set of program outputs some suitable criterion of program quality can be defined by annotations, and every concretizing transformation of an annotated program is aimed at improving the program according to the qualitative criterion without disturbing the meaning of the program in the application context defined by annotations.

```
forward function fact (n: integer
    /*$ assert=n>=1*/
    /*$ assert=_>=n*/
    returns integer)
function fact (n: integer returns integer)
  if n = 1 then 1
  else /*$ assert = _ > 0 */
     fact(n-1)*n
  end if
end function
```

Fig. 5. Cloud-Sisal-program with optimizing annotations.

According to the approach used [10], any source program is considered as a base for constructions of a number of different specialized programs. Every construction starts with the source program and an application context conveyed in annotations. Some program annotations can be formed in parallel with the development of the source program, others are added by users and describe a specific context of source program applications. Then a series of concretizing transformations is applied to the annotated general-purpose program (either automatically or interactively with the user), which results in a correct and qualitative specialized program.

For example, every expression in Cloud-Sisal-program can be prefixed by a pragma "assert = Boolean expression", that can be checked for truth after the expression evaluation during program debugging and then can be used in program optimizing transformations. The result of the expression can be denoted as the underscore symbol "_" and if the expression is n-ary (where $n > 1$), then its components can be denoted as an array with the name "_": "_[1]",..., "_[n]". In addition, the pragma "assert = Boolean expression" can be placed before returns keyword in procedure declarations and can be used to control results of this procedure after its invocation. As an example of the assert pragma statement usage please consider factorial function declaration and definition which

are represented in Fig. 5. Another example of optimizing annotations is a pragma "parallel" which can be used before a case expression in Sisal (analogous to a switch expression in C language). This pragma can be specified if it is known that only one test can be true. The pragma of the form "parallel = Boolean expression" means that only one test is true if the specified Boolean expression is true.

4 Internal Representations

The CSS system uses three internal presentations of Cloud-Sisal-programs: IR1, IR2 and IR3.

```
function sign (N: integer returns integer)
 if N > 0 then 1
 elseif N < 0 then -1 else 0 end if
end function
```

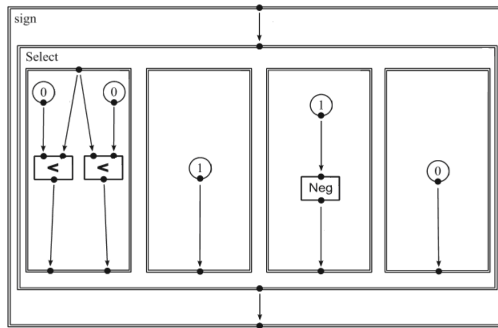

Fig. 6. A function sign and its IR1-representation.

4.1 Representation IR1

IR1 is a language of hierarchical graphs [12] made up simple and compound computation nodes, edges, ports and types (See Fig. 6). Nodes correspond to computations. Simple nodes are vertices and denote operations such as add or divide. Compound nodes are subgraphs and represent compound constructions such as structured expressions and loops. Ports are vertices that are used for input values and results of compound nodes. Edges show the transmission of data between simple nodes and ports; types are associated with the data transmitted on edges. IR1-program represents data dependencies, with control left implicit; e. g. iteration is represented as a compound node with subgraphs describing generation of index values, the body of the loop, and the packaging of results.

4.2 Representation IR2

IR2 is an extension of IR1. All nodes in the IR2 graph are partially ordered by the $\preceq_e$ ordering in such a way that if $N_1 \prec_e N_2$, then N_1 must be executed before N_2, and if $N_1 =_e N_2$, then N_1 and N_2 can be executed in any order, and a parallel execution is possible. All edges in the IR2 graph are decorated by variables (See Fig. 7) which will be the operands of IR3 operations. Each variable has the following attributes: a unique identifier, a unique name, a type and an additional Boolean variable which defines the "IsError" property. The types in IR2 and IR3 represent the types of the Cloud Sisal language within IR2 and IR3. Each type contains additional low-level information about objects (such as machine representation of the type). IR2 is intended to provide a natural and usable structure for optimizations. During the optimization process, the optimizations can create additional data connected with a node, an edge or a port. The data created by one optimization can be reused by another.

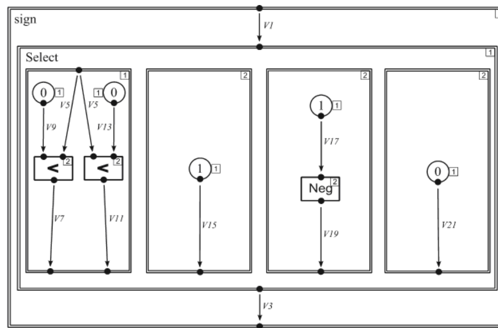

Fig. 7. IR2-representation of the function sign.

4.3 Representation IR3

IR3 is a classical three-address code representation with hierarchical blocks. For example, function sign can be represented as follows:

```
0 entry "function sign[integer]" (V_1(I32) returns V_3(I32));
   {
1    V_5(I32) = V_1(I32);
2    V_5(I32) = V_1(I32);
3    V_9(I32) = 0x0(I32);
4    V_13(I32) = 0x0(I32);
5    V_7(BOOL) = (V_9(I32) < V_5(I32));
6    V_11(BOOL) = (V_5(I32) < V_13(I32));
7    if (V_7(BOOL) == true(BOOL))
```

```
     {
10     V_15(I32) = 0x1(I32);
11     V_3(I32) = V_15(I32);
      }
     else
      {
12     if (V_11(BOOL) == true(BOOL))
        {
15       V_19(I32) = 0x1(I32);
16       V_17(I32) = - V_19(I32);
17       V_3(I32) = V_17(I32);
         }
       else
       {
18      V_21(I32) = 0x0(I32);
19      V_3(I32) = V_21(I32);
       }
      }
20  return;
   }
```

5 Cross-compiler

The optimizing cross-compiler of the CSS system consists of two main parts: front-end and back-end compilers (Fig. 8).

The front-end compiler translates Cloud-Sisal-modules into a monolithic IR1-program which is used also by the interpreter and the graphic visualization/debugging subsystem.

The back-end compiler begins with R2Gen which produces a semantically equivalent program in IR2. Then the IR2Opt subsystem performs some optimizations and concretizations on the annotated program to produce a semantically equivalent, but faster basic program. After completion of the machine-independent optimizations, the IR3Gen subsystem preallocates array storage where compile time analysis or compiler generated expressions executed at run time can calculate the final size of an array. The result of this phase is the production of a semantically equivalent program in IR3. The next phase of compilation (IR3Opt) performs update-in-place analysis and restructures some graphs to help identify at compile tune those operations that can execute in-place and to improve chances for in-place operation at run time when analysis fails. It performs also some machine-dependent optimizations and defines the desired granularity of parallelism based on an estimate of computational cost and various parameters that tune analysis. After parallelization, CodeGen generates C++ or C# code, and the compilation can be completed using the target machine's C++ or C# compiler.

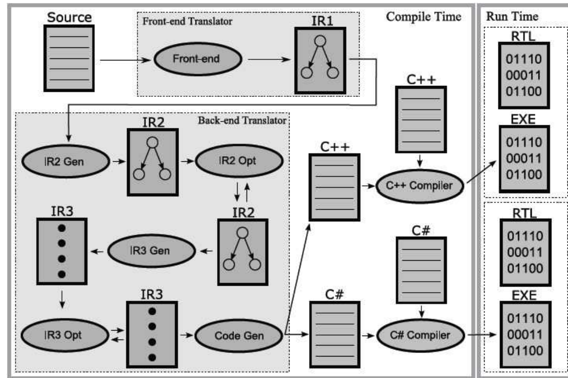

Fig. 8. The Cloud-Sisal-compiler and run-time support.

6 Related Works

New parallel language development is not popular today; more popular is existing language extension (sometimes it is positioned as a separate language); such approach keeps sequential semantics problems, but considered as the fastest both for the developer and for the final application execution. In this section we will not observe such extensions as related.

6.1 The Pifagor language

This language has been developed as a functional language for creation of architecture-independent parallel programs [17]. The Pifagor language is optimized to dataflow graph description and has a specific syntax which is not easy to understand because it differs from common imperative and functional languages. For example, the follow-ing Pifagor function performs vector multiplication by scalar:

```
VecScalMult << funcdef Param
// Argument format: ((x1, x2, : xn), y),
// where ((x1,x2,:xn) is a vector, y is a scalar
{
((Param:1,(Param:2,Param:1:|):dup):#:[]:*) >>return
}
```

It is hard to compare Pifagor syntax and constructions with Sisal because they are completely different. Sisal has loops and arrays; we suppose it is better for science computational tasks. According to the articles of the Pifagor developers [17] the Pifagor language is aimed on the list processing and the conception of unlimited parallelism scheduled as limited at runtime. At present, there are both an experimental compiler and an experimental interpreter for the Pifagor language that are used for scientific proposes such as a development of the new scheduling algorithms and for parallel programming education.

6.2 The F# language

We can't say that F# [18] is the project in a same direction with Sisal, but Microsoft's developments in a functional paradigm can't be avoidable. As the complexity of the systems was increased the complexity of compiler grows and some features of the functional languages formerly considered as ineffective started to implement in imperative languages.

On the one hand F# is functional ML-family language and functional paradigm suits better for parallel computations. On the other hand F# has an ability to create any mutable indexes, non-functional calls or dependencies, external .NET objects and operations. It can't be considered as single assignment or parallel language. It is hybrid language that can be used for writing both implicitly parallel and sequential programs. Multi-threaded programming on F# is quite similar to C# or C programming.

Not only in case of the F# language but for the most of functional languages developers are trying to make the programming language available for wide range of users but it makes language less pure and less functional. State modification operators such as input and output give the developer familiar ability to process the data but make the semantic sequential or non-deterministic.

7 Conclusion

The project of visual cloud system CSS aimed at supporting of functional and parallel programming teaching and learning was considered.

The CSS system is intended to provide means to write and debug functional programs regardless target architectures on low-cost devices as well as to translate them into optimized parallel programs, appropriate to the target execution platforms, and then execute them in clouds on high performance parallel computers without extensive rewriting and debugging. So, the CSS system can open the world of parallel and functional programming to all students and scientists without requiring a large investment in new, top-end computer systems.

At present, the CSS system consists of experimental versions of Web-interface, interpreter, graphic visualization / debugging subsystem, optimizing cross-compiler. The current target platform for the Cloud-Sisal-compiler is .NET. The compiler performs conventional optimizing transformations and generates the C# code. It allows the users to perform the experimental execution of Cloud-Sisal-programs and examine the effectiveness of optimizing transformations applied by the compiler. We starts some experiments of using our system for teaching and leaning of functional and parallel programming as well as of optimizing compilation and high performance computing.

Acknowledgments. Our thanks to all colleagues taking part in the project described. This research is supported by the Russian Foundation for Basic Research under grant RFBR N 15-07-02029.

References

1. Backus, J.: Can programming be liberated from the von Neumann style? Commun. ACM. 21, 8, 613-641 (1978)
2. Cann, D. C.: Retire Fortran?: a debate rekindled. Commun. ACM. 34, 8, 81-89 (1992)
3. Gaudiot, J.-L., DeBoni, T., Feo, J., et all: The Sisal project: real world functional programming. In: Pande, S., Agrawal, D.P. (eds.) Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems. LNCS, vol.1808, pp. 45-72. Springer, Heidelberg (2001)
4. McGraw, J., Skedzielewski, S., Allan, S., Grit, D., Oldehoeft, R., Glauert, J., Dobes, I., and Hohensee, P.: SISAL-Streams and Iterations in a Single Assignment Language, Language Reference Manual: Version 1.2. Technical Report TR M-146, University of California, Lawrence Livermore Laboratory, March (1985)
5. Feo, J. T., Miller, P. J., Skedzielewski, S. K., Denton, S. M., Solomon, C. J.: SISAL 90. In: Proceedings of High Performance Functional Computing, pp. 35-47, Denver (1995)
6. Kasyanov, V. N., Stasenko, A.P.: Sisal 3.2 programming language. In: Tools and techniques of program construction, pp. 56-134, Novosibirsk (2007) (In Russian)
7. Kasyanov, V. N.: Sisal 3.2: functional language for scientific parallel programming. Enterprise Information Systems. 7, 2, 227-236 (2013)
8. Cann, D. C., Feo, J.T., Bohm, A. P. W., et all: Sisal Reference Manual: Language Version 2.0. Tech. Rep. Lawrence Livermore National Laboratory, UCRL-MA-109098, Livermore, CA (1991)
9. Kasyanov, V. N. Idrisov R.I., Kasyanova E.V., Stasenko A.P.: A parallel programming language Cloud Sisal. In: Proceedings of XVI International Conference “Informatics: problems, methodology, technology”. vol. 5, pp. 157-161, Voronezh (2016) (In Russian)
10. Kasyanov, V. N.: Transformational approach to program concretization. Theoretical Computer Science. 90, 1, 37-46 (1991)
11. Kasyanov, V. N.: A support tool for annotated program manipulation. In: Proceedings of Fifth European Conference on Software Maintenance and Reengineering, pp. 85-94, IEEE Computer Society Press, (2001)
12. Kasyanov, V. N., Kasyanova, E. V.: Information visualization based on graph models. Enterprise Information Systems. 7, 2, 187-197 (2013)
13. Kasyanov, V. N., Kasyanova, E. V.: Graph- and cloud-based tools for computer science education. In: Boumerdassi, S., Bouzefrane, S., Renault, E. (eds.) MSPN 2015. LNCS, vol. 9395, 41-54. Springer, Heidelberg (2015)
14. Gordeev, D.S.: Visualization of internal representation of Cloud Sisal programs. Scientific Visualization. 8, 2, 98-106 (2016) (In Russian)
15. Idrisov, R.: Sisal: parallel language development. In: Proceedings of the 6th Spring/Summer Young Researchers Colloquium on Software Engineering (SYRCoSE 2012), pp. 38-42, Perm (2012)
16. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, K.: Efficiently computing static single assignment form. Transactions on Programming Languages and Systems. 13, 4, 451-490 (1991)
17. Legalov, L.: Functional language for creation of architecture-independent parallel programs. Computational Technologies. 10, 1, 71-89 (2005) (In Russian).
18. Syme, D., Granicz, A., Cisternino, A.: Expert F# 3.0. Apress (2012)