A System of Functional Programming for Supporting of Cloud Supercomputing

VICTOR KASYANOV, ELENA KASYANOVA
Institute of Informatics Systems
Novosibirsk State University
Novosibirsk, 630090
RUSSIA
kev@iis.nsk.su

Abstract: - In this paper, a cloud system being under development at the Institute of Informatics Systems in Novosibirsk as a system of functional programming for supporting of cloud supercomputing is considered. The 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 without extensive rewriting and debugging.

Key-Words: - Cloud supercomputing; functional programming; optimizing cross compilation, parallel programming.

1 Introduction

Parallel computing is one of the main paradigms of modern programming and covers an extremely wide range of programming issues. In view of the much more complex nature of parallel computations in comparison with successive ones, the methods of automating the development of parallel software, based on applying the technique of formal models, specifications and transformations of parallel programs, are of great importance.

The fundamental problems of organizing parallel computing are the following: the problem of increasing the productivity and efficiency of using multiprocessor and distributed computing systems and the problem of increasing the level of intellectualization of programming parallel systems. They are not independent, because the organization of high-performance computing in a multiprocessor system of modern architecture is too complex for attempts to solve it without the means of intellectualization of programming in such a system. The difficulty in solving the problems of programming parallel systems is determined by the fact that the issues of organizing interactions and synchronizing parallel processes significantly complicate the development of parallel algorithms and programs in comparison with their traditional (sequential) versions.

One of the most promising ways to solve these problems jointly is the development of declarative means of describing and implementing parallel computations.

The first language of functional programming was Lisp, developed in the late 1950s. by the American scientist John McCarthy. Although the language was widely known, due to its greater expressiveness and elegance compared with traditional languages, its applicability was limited mainly to the tasks of artificial intelligence. A new period of functional programming began with the 1978 Turing lecture of John Backus [1]. This new understanding and wider acceptance of functional programming was determined, first of all, by the process begun in those years to move to the consideration of the programming problem in its full context, beginning with the specification of the problem and the logical analysis of its solvability, the byproduct of which is the program itself. The emergence of computational systems with parallel architectures further increased the importance of functional programming, as it allows the user to free himself from most of the parallel programming problems inherent in imperative languages and to entrust the compiler with the construction of a program effectively executed on a computing system of a particular parallel architecture. In addition, many technical problems of system and application programming become clear when presenting their solutions in a functional style.

Therefore, it is no coincidence that the design work in the field of designing new technical means

is now often carried out with the help of CAD-systems, usually organized in a functional style. The most complex new tasks of using computers in non-traditional areas, as well as in linguistic, chemical, biological, medical and other intellectualized spheres of activity are often solved first on functional or logical programming languages, and these solutions are used as the base for finding solutions at the level of standard imperative languages. There is gradually increasing the number of universities that choose functional programming languages as a conceptual basis for teaching computer science. All this makes us predict the further spread of the functional approach, especially in the intellectual and knowledge-intensive fields of application of modern technology. However, the successful development of the functional approach to supercomputing raises a number of problems, the study of which has been occupied by many teams in recent years. The project described in this paper is aimed at solving these problems, and within the framework of the approach approved in the late 70's in the languages VAL and BARS and successfully developed in a number of DCF projects, Pythagoras, COLAMO, etc., among which the SISAL language [2], the first version of which refers to 1983.

The tendency of the development of programming is that more and more diverse processes of processing programs and data and are increasingly supported by the machine. Most of these processes for processing programs and data are implemented in existing tools as text or language, but are semantic. In them, as a rule, it is required to preserve a certain invariant that is definitely associated with the semantics of the processed objects (for example, translation and other functionally equivalent transformations of programs preserve the function implemented by the program). Therefore, without comprehensive study and profound use of semantic transformations in instrumental systems, it is impossible to achieve either reliable or effective solution of the problems of programming automation, to switch from handicraft production of programs to technology and mass production.

The transformational approach [3] treats programming as a systematic application of the fundamental processes of semantic processing of programs that preserve a certain semantic invariant of the program and make up “a sum of technologies” in the aggregate. Transformational methods are used as the main means for achieving efficiency in the automation of programming by translation methods, especially in connection with the advent of computers of new architectures. They are a promising direction in creating new, more powerful means of automating the design of effective and reliable programs.

Works on the theoretical substantiation of the transformational approach to software development are actively developing all over the world. At the same time, researchers still have the task of developing an “algebra of programs” that allows manipulating program fragments within the formal calculus of programs in order to automate the construction of effective and reliable programs for advanced computing systems. This very tempting goal is unlikely to be achieved in the near future due to the variety of programming languages and specifications used, as well as the architecture of the computing systems. However, if the functional level for the initial specification is fixed, then the development of methods and technologies for transforming this specification into a correct and efficient program for computers with different architectures can be considered a realistic task.

In this paper, we describe the CSS (Cloud Sisal System) system which is under development at Institute of Informatics Systems in Novosibirsk as a system of functional programming for supporting of cloud supercomputing.

The rest of the paper is structured as follows. Section 2 gives a general description of the CCS system. Section 3 describes its input language (Cloud Sisal). In Section 4 the Cloud Sisal compiler of the SSC system is outlined. Section 5 is our conclusion.

2 The CCS System

The CSS system is aimed to be a general-purpose user interface for a wide range of parallel processing platforms (See Fig. 1). The input language of the CSS system is a functional language Cloud Sisal that exposes implicit parallelism through data dependence and guarantees determinate result. In our conception, the cloud interface is intended to give transparent ability to execute functional programs on arbitrary environments. The JavaScript client does not demand installation, and small functional programs can be executed on client devices (computers or smart phones). The V8 server allows the language parser and some optimizations to be used at both client and server sides.

SISAL (Steams and Iterations in a Single Assignment Language) is developed as a functional programming language, specifically oriented to parallel processing and the replacement of the Fortran language on supercomputers in scientific computing [5], [6]. It is still early to speak about

real displacement, but SISAL as a parallel programming language for scientific computing is quite interesting itself and has already found its application in dozens of organizations around the world.

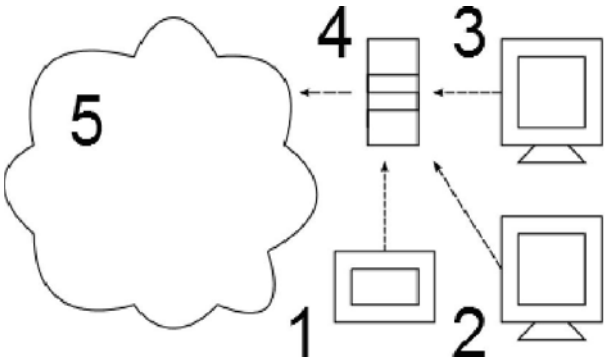

Fig. 1. Cloud service structure: 1, 2 and 3 – clients, 4 – cloud access server, 5 – execution environment.

The SISAL model uses what is called an implicit parallel model. This model places more control in the hands of the compiler that actually encodes a program into a particular machine language. The details of parallel resource management are handled by the reliably and portably by a compiler and runtime system rather than by an error prone human programmer. The language's functional semantics guarantee determinate results across parallel and sequential implementations - something that is impossible for traditional languages like Fortran to guarantee. Moreover, SISAL's implicit parallelism makes it unnecessary to rewrite any code to move from one computer to another. A SISAL program that runs correctly on a PC is guaranteed to run correctly on a high-speed parallel computer.

There are several implementations of SISAL (version 1.2 [2]) for supercomputers, in particular Denelcor HEP, Vax 11-780, Cray-1, Cray-X / MP, work is under way to create a prototype optimizing compiler from SISAL 1.2 in distributed programs for calculators, hardware or software supporting multithread computing, such as, for example, TERA, *T, TAM and MIDC.

The Livermore National Laboratory and Manchester University have developed an improved version of the SISAL-90 language [7], which increases the language's utility for scientific programming. It includes 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 [8] integrates features of Sisal 2.0 [9] and SISAL-90 versions and includes language level support for module design, mixed language programming, and preprocessing.

The Cloud Sisal language that has been designed as the input language of the CSS system is based on the Sisal 3.2 and increases the language's utility for supporting of scientific computations and parallel programming in clouds (See Section 3).

The CSS system is intended to provide means to write and debug Cloud-Sisal-programs regardless target architectures on low-cost devices as well as to translate the Cloud-Sisal-programs into optimized imperative parallel programs, appropriate to the target execution platforms, and then to execute them on supercomputers in clouds. The CSS system includes for Cloud Sisal programs both an interpreter and an optimizing cross-compiler. It contains also some usual parts like syntax highlighting, persistent storage for program code, authorization and so on.

The CSS system uses three internal presentations of source Cloud-Sisal-programs (See Section 4). The first internal representation (IR1) is used by both the interpreter and the compiler. IR1 is a language of so-called hierarchical graphs [10] made up simple and compositional nodes, edges, ports and types. Nodes correspond to computations. Simple nodes denote operations such as add or divide. Compositional nodes represent compound constructions such as structured expressions and loops. Ports are nodes 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. So, IR1-program represents data dependencies, with control left implicit.

The interpreter is available on web via a browser. It translates a source Cloud-Sisal-program to its hierarchical graph representation (IR1-program) and runs it without making actual low-level code. It is useful because in this case a user can get any debugging information both in text and in visual form of hierarchical graphs.

The optimizing cross-compiler is intended to convert a well-functioning (debugged) Cloud-Sisal-program into optimized imperative parallel programs, appropriate to the target execution platforms (See Section 4).

The compiler generates also a GraphML-file with a graph which represents data structures handled by the compiler. GraphML (or Graph Markup Language [11]) is at present de facto standard language for describing graphs. GraphML is XML sublanguage and allows describing directed, undirected, mixed, hyper, and hierarchical graphs as well as different attributes of their elements. It is assumed that this file generated by the cross-compiler can be used by a user for post-mortem

visualization with the help of the Visual Graph system [12]. The Visual Graph system can be used to read this graph from the GraphML-file, to visualize it and to provide a user with different navigation tools for its visual exploration to take the most optimal decisions.

3 Cloud Sisal Language

A Cloud Sisal program consists of one or more separately compilation units called modules. Each module consists of definition and declaration files and contains definition and declaration of procedures (functions and operations), types and contract definitions.

A Cloud Sisal module declaration contains procedure declarations which are defined by the corresponding module definition and are visible outside it. It contains also externally visible declarations and definitions of types and contracts. A Cloud Sisal module declaration may specify the name of a record or union type for public use, but may prevent exportation of the components. It is assumed that any function in any module may be the starting point of program execution. At this outermost level, all function parameters are values obtained from the operating system level and all function results are produced at that level.

Since Cloud-Sisal-compilers can translate Cloud-Sisal-programs into the C programming language, all Cloud Sisal function definitions have corresponding C language equivalent definitions which in turn have corresponding declaration in the C language which allows program not written in Cloud Sisal to have subsidiary parts written in the Cloud Sisal language. Special foreign module declarations declare the relationship between Cloud Sisal and a set of subsidiary code written in other languages. This allows Cloud Sisal program to access libraries of already written code.

Data types of the Cloud Sisal language include the usual scalar types (boolean, character, integer, real, double), structured types (records and unions, arrays and streams) and functions.

Structured types may have values of any type as components; records and unions have heterogeneous components and arrays and streams have homogeneous components. The Cloud Sisal language supports also user defined types with their custom operations.

Function values may be parameters to functions and the results of expression evaluation, so function types may be declared by giving the types of all parameters and results. Therefore the Cloud Sisal language does not use a complete type inference system wherein the types of all values are inferred from their contexts. As a result complete compile-time typing is possible for all Cloud-Sisal-programs.

A Cloud Sisal function is declared by describing its name, the names and types of its formal parameters, and types of its result values. The function contents one or more expressions (a multi-expression) whose types correspond to the result types. Values are available to the expressions via formal parameters, not through globally accessed names. Higher-order function operations are part of the Cloud Sisal language. Functions can be passed to and returned from functions and be the values of expressions.

Expressions are, of course, the heart of the Cloud Sisal language. Syntax is designed to be as familiar to more traditional procedural languages (like Pascal) as possible. Conventional infix operations combine scalar arithmetic values. The Cloud Sisal language supports some type promotion automatically and provides some predefined type conversion functions. It is possible to assign the value of any expression to a name and use the name as shorthand for the expression throughout the scope of the definition. This scoping is done with the let construct.

All Cloud Sisal expressions, including whole functions, programs and loops, evaluate to value sets. In Cloud Sisal programs it is possible to use three kinds of loops: post-conditional, pre-conditional and “for all” (operation is applied to a set). So-called reductions can be used to determine returning values of loops. Keyword “returns” at the end of a loop is followed by name and parameters of a reduction. 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.

In the Cloud Sisal language “always finished computations” semantics is used. It means that an execution will not stop on any error and will return resulting value even if an error occurs. Each Cloud Sisal type has a distinguished value, “error”. Any failed expression evaluation results in “error” of the appropriate type. The “error” values propagate in a well-defined way when they are operands in computations. The “error” values can be tested for and even explicitly assigned to signify other anomalous conditions.

Cloud Sisal language was designed to describe scientific computations so after analysis of features of other languages with scientific orientation (such as the Fortran language) it was decided to introduce multidimensional arrays and arrays with fixed form to the Cloud Sisal language as well as extended

means for their construction. Cloud Sisal has comprehensive facilities for defining and manipulating array values. An array generator allows the definition of a multidimensional object whose parts form a "tiling" of the overall structure. Selections of arbitrary subarrays are provided beyond the rectangular subsets available in some other notations. Many infix operations for operating element-by-element on array operands and a useful set of functions on arrays are defined. A subarray update facility allows safe alteration of array values. Many applications can be expressible succinctly with these features. Array generation, selection and update may use vector subscripts to refer to arbitrary, non geometric sections of arrays.

A stream is a sequence of values produced in order by one expression evaluation and consumed in the same order by one or more other expression evaluations. Producers and consumers are usually for expressions but short forms for simple streams are also available. To expose the pipelined parallelism that streams make possible, they must be implemented non-strictly. That is consumer expressions must be started whether or not the producer expression has finished.

The Cloud Sisal includes support of functions from programs and libraries written in C/C++ or Fortran. It is based on a concept of so-called foreign types. Foreign types in the Cloud Sisal language are specified by their string representation on their native programming language. Values of foreign types are constructed via foreign operations and functions that are written in a foreign programming language and use a special interface to access values of the Cloud Sisal types if necessary.

To increase level of its algorithmic abstractions, the Cloud Sisal language was augmented by new conceptions of parametric types, contracts and generalized procedures (functions and operations). A parametric type defines a set of types that allows finer control compared to already existing typesets. A contract is another form of abstraction that allows binding a set of operations over types listed as contract parameters to contract name. Contracts are used in generalized procedures to specify what kind of operations their parametric types are expected to have.

It should be noted that a user-defined reductions of the Sisal 90 language are functions of a very special form that are used to transform loop values into loop results and cannot be reused outside loops. In the Cloud Sisal language user-defined reductions are defined as a combination of several usual functions thus allowing them to be reused. A general form of reduction invocation in a loop return statement looks as follows: "reduction name *N* (list of initial values) of (list of loop values)" where initial values of reduction must be loop constants and reduction name *N* corresponds to following four functions. The first function computes an initial reduction state in a type *T* which is any type that can hold a reduction internal state. The second function recomputes the reduction state *T* using loop values of the subsequent loop iteration. The next optionally present function determines how some two reduction states (obtained after parallel loop execution) can be merged. This function can be omitted if the reduction does not allow such things. The last function computes reduction results from its internal state.

The Cloud Sisal language supports also so-called annotated programming and concretizing transformations [13], [14] and specifies optimizing annotations in a form of so-called pragma statements. Every pragma statement is a formalized comment that starts with dollar sign "$" and describes an annotation being a predicate constraint on admissible properties of a program fragment or states of computations.

For example, every expression of Cloud-Sisal-program can be prefixed by an annotation

"assert = Boolean expression",

that can be checked for truth after the expression evaluation during program debugging as well as can be used in program optimizing and concretizing transformations. This annotation can be also placed before returns keyword in declaration of a procedure and can be used to control results of this procedure after every its invocation. Another example is an annotation "parallel" which can be placed before any case expression in the Cloud Sisal program (analogous to a switch expression in the C language). This annotation can be specified if it is known that only one test can be true.

3 Optimizing Cross-Compiler

The general scheme of the cross-compiler of the CSS system is shown in Fig. 2 The compiler consists of two main parts (front-end and back-end compilers), and the compilation can be completed using the target machine's C++ or C# compiler.

The original Cloud Sisal program ("Source" in the diagram) is fed to the input of the front-end translator, which translates it into the first internal representation (IR1). Further the graph IR1 is fed to the input of the back-end part of the compiler.

The back-end part includes the following phases (where optimization phases are optional):

- translation from IR1 to IR2 ("IR2 Gen" in the diagram) which produces a semantically equivalent program in IR2,
- IR2 optimization ("IR2 Opt" in the diagram) which performs some optimizations and concretizations on the annotated program to produce a semantically equivalent, but faster basic program,
- translation from IR2 to IR3 with generation of parallel code ("IR3 Gen" in the diagram),
- IR3 optimization and IR3 level reduction ("IR3 Opt" in the diagram) which 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, translation from IR3 into the code of the target architecture ("CodeGen" in the diagram) which generates C++ or C# code.

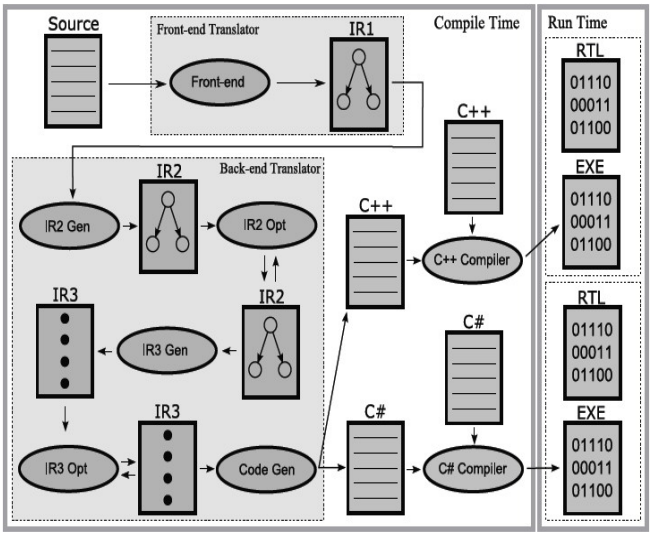

Fig. 2. The Cloud-Sisal-compiler and run-time support.

The IR-representation of a Cloud-Sisal-program consists of a set of directed acyclic graphs IR1 corresponding to the functions of the source Cloud-Sisal-program.

The IR-graph of a function is given by the triple $G = (N, P, E, P^{in}, P^{out})$, where

- N is a set of nodes,
- P is a set of ports,
- $E \subseteq P \times P$ is a set of arcs,
- $P^{in} \subseteq P$ is a set of input ports of the graph G,
- $P^{out} \subseteq P$ is a set of output ports of the graph G.

Each node $N_i \in N$ corresponds to two subsets of ports from P: input ports $P_i^{in} \subseteq P$ and output ports $P_i^{out} \subseteq P$. The arc $E_i=(P_1, P_2) \in E$, if for some i and j either $P_1 \in P_i^{out}$ and $P_2 \in P_j^{in} \cup P^{out}$ or $P_1 \in P_{in}$ and $P_2 \in P_j^{in} \cup P^{out}$.

The nodes of graph IR1 specify operations, ports specify arguments and results of operations, and arcs specify information dependencies between them. Thus, the graph IR1 specifies the flow of computations. The graph IR1 is a hierarchical graph [10], since the set of vertices of N includes nodes of two types: simple and compositional. Compositional nodes contain the child graphs IR1, the dependencies between the ports of which are expressed implicitly, and denote structured constructions of the Cloud Sisal language, such as conditional expressions and loops. Each arc of graph IR1 is assigned the type of the value sent by it, if IR1 specifies a strongly typed language. Fig. 4 shows[1] the graph IR1 specifying the function "sort" represented in Fig. 3.

```
function sort(A: array[integer]
              returns array[integer])
 if size(A) <= 1 then A
   //not necessary to order everything
 else let /* otherwise we return
   the result of the expression "let",
   which associates names with arrays
   of values which are less, equel and
   lardge numbers of the number A[1]
   (numbers are selected from
   the array A) */
      Less := for a in A
           returns array of a
           when a < A[1] end for;
      Same := for a in A
           returns array of a
           when a = A[1] end for;
      More := for a in A
           returns array of a
           when a > A[1] end for
      /* the result of the expression
      is concatenation (||)
      of three arrays */
    in sort(Less)||Same||sort(More)
 end let end if
end function
```

Fig. 3. The Cloud-function "sort".

[1] The drawings of the IR1 graphs are generated automatically by the internal representation of IR1, built by the front-end compiler.

In IR1, conditional expressions are defined by a compositional node of Select, whose first graph is the control. The control graph has one output of an integer type, whose value determines which of the graphs of the compositional node Select (starting with the second graph) will be used to generate its output values. Fig. 5 shows the contents of the composite node Select represented in Fig. 4. Fig. 6, 7 and 8 depict the contents of the first, second and third graphs of the compositional node Select, respectively. The literal in the graph IR1 is given by nodes without input ports and with one output port. Since IR1 is described by a hierarchy of acyclic graphs, the function calls in Fig. 8 are given by a functional-type literal with a function name that uniquely identifies its graph.

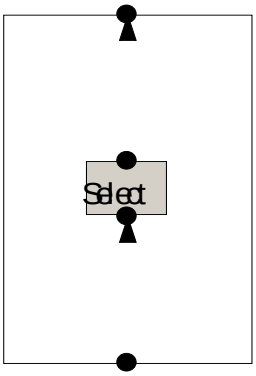

Fig. 4. The graph of the function “sort”.

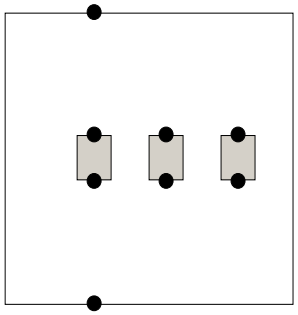

Fig. 5. The graph of the composite node Select.

In the IR1 graphs, cyclic expressions are specified by the compositional nodes LoopPost, LoopPre, and Forall, which specify loops with a postcondition, precondition, and range, respectively. These compositional nodes have a constant number of graphs which describe the control of the loop, the body of the loop, and its return clause. Fig. 9 shows the contents of the composite Forall node in Fig. 8 (corresponding to the array “Same”). In Fig. 10 and Fig. 11 shows the contents of the second (specifying the generation of the range) and the fourth (specifying the return proposition) graph of the composite Forall node, respectively (the first and third parts are empty).

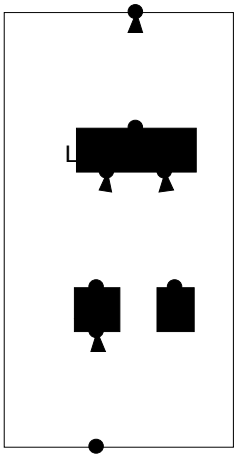

Fig. 6. The first graph of the node Select.

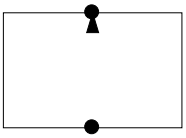

Fig. 7. The second graph of the node Select.

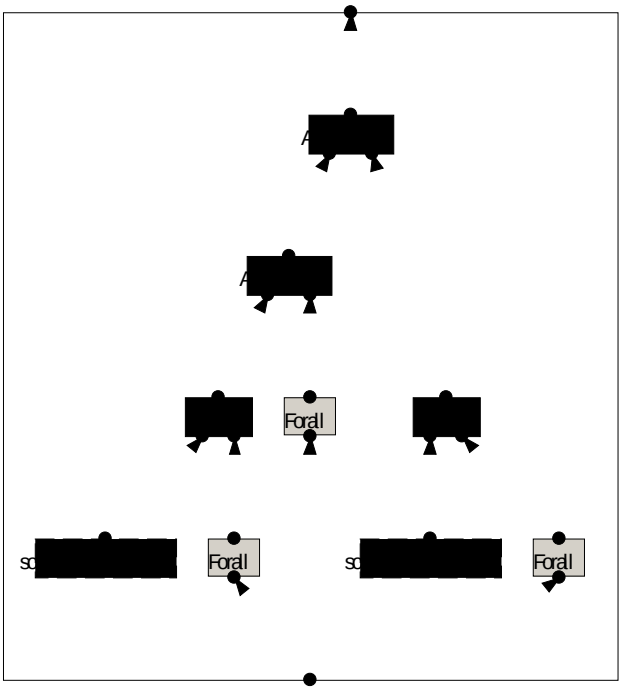

Fig. 8. The third graph of the node Select.

The internal representation of IR2 is based on IR1. A graph IR2 for a Cloud Sisal function is an object $(G, VAR, \sigma_\upsilon, \preceq_\beta)$, where

- $G = (N, P, E, P^{in}, P^{out})$ is a graph IR1,
- *VAR* is a set of variables,

- σ_υ is a map $E \to VAR$ which specifies the binding of variables to the arcs of the graph G,
- $\leq_\beta$ is an order on $N \times N$ specifying the execution sequence.

All vertices of the graph IR2 are ordered by the relation $\leq_\beta$, which is constructed from the data stream. Namely, if there exists a path from node N_1 to node N_2, where N_1 and N_2 have the same nesting level, then $N_1 \leq_\beta N_2$. For nodes N_1 and N_2 that are contained in the same composite node (such as conditional expressions and loops), the relation of the execution sequence is determined by the semantics of the composite node. If N_1 and N_2 are not contained in the same composite node and there are no restrictions imposed by the rules of the composite nodes, $N_1 =_\beta N_2$[2] is assumed.

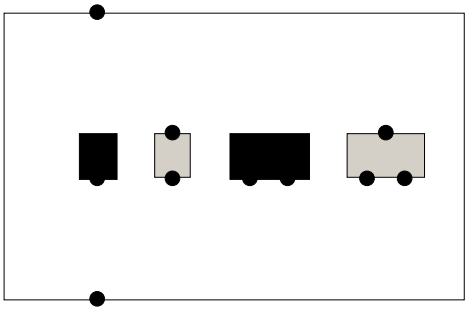

Fig. 9. The graph of the composite node Forall.

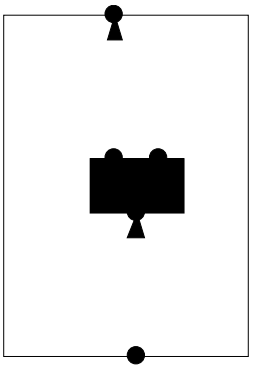

Fig. 10. The second graph of the node Forall.

For internal representations of IR2 (and IR3), variable and type objects are defined. The type in IR2 (and IR3) serves to represent the types of Sisal language at the back-end level of the translator. A type contains low-level information about the object with which this type is associated (such as a machine type representation and a memory class).

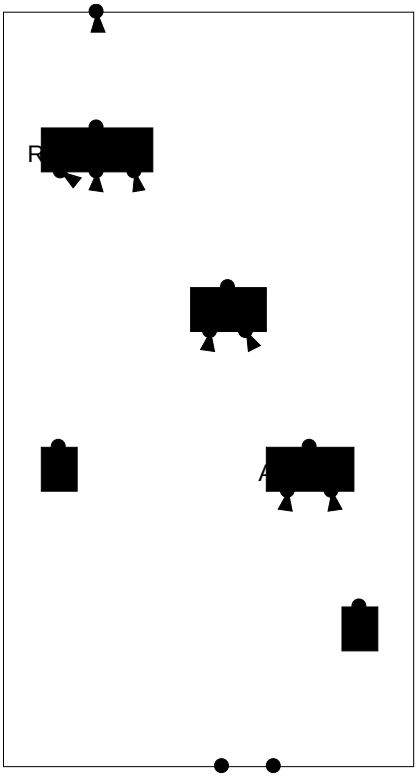

Fig. 1. The fourth graph of the node Forall.

The variables describe objects of the Cloud Sisal language at the level of IR2 and IR3. In the IR2 representation, the variables are associated with the graph arcs (in IR3, they are the operands of the IR3 operations). Each variable has the following properties: a unique identifier, a unique name, a type and an optional Boolean variable that is responsible for the "is error" property. Variables are divided into scalars, arrays and records. Each group of variables has some additional properties. Scalar variables additionally have size in bytes. Array variables additionally contain three auxiliary variables: a variable that describes the array element, the lower bound of the array, and its size. A record variable contains a list of variables describing the fields of this record.

The IR2 representation is built for the translated module and is the set of IR2 graphs for the functions contained in this module. Fig. 12 shows the representation of the IR2 function of the following Cloud Sisal function which calculates the sign of the number:

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

[2] The relation $\leq_\beta$ is used to fix the sequence in which the operators represented by nodes of IR2 must be executed. If $N_1 \leq_\beta N_2$ then the computation of N_1 must necessarily occur before calculating N_2. If $N_1 =_\beta N_2$ then it is possible to perform operations for these nodes in any order, and so they can be executed in parallel.

```
   end if
end function
```

From the point of view of the structure, the graph IR2 which is obtained immediately after translation from IR1 to IR2 is isomorphic to the graph IR1, and therefore, the actual creation of nodes and arcs of the representation IR2 is trivial. After the graph is created, the mapping σ_v is constructed via annotating arcs of graph IR2 by variables. Arcs coming from the same port are assigned the same variable. Arcs coming from different ports have different variables. This achieves the requirement that the definition of each variable be unique. Later (after the optimizations of the IR2 presentation), the distribution of variables along the arcs of the graph can change. The final step in the construction of the IR2 presentation is the ordering of the vertices by the priority-of-execution relation $\leq_\beta$.

The IR3 representation is a mid-level imperative representation of the program, consisting of statements and expressions. It is a classical three-address code representation with hierarchical blocks. In the process of translation from IR2 to IR3, an imperative program (a sequence of operators) is actually constructed for the graph of the IR2 calculation data flow, performing the calculations given by this graph. For example, the sign function can be represented as follows:

```
  0   entry "function sign[integer]"
(V_1(I32)  returns V_3(I32)); {
  5      V_7(BOOL) = (0x0(I32) <
V_1(I32));
  6      V_11(BOOL) = (V_1(I32) <
0x0(I32));
  7      if (V_7(BOOL) == true(BOOL))
{
 11         V_3(I32) = 0x1(I32);
          } else {
 12         if (V_11(BOOL) ==
true(BOOL)) {
 17          V_3(I32) = - 0x1(I32);
            } else {
 19          V_3(I32) = 0x0(I32);
            }
          }
 20      return;
        }
```

The construction of IR3 from the graph IR2 involves generating a sequence of operators for each node and placing the resulting fragment in the general sequence of IR3 operators.

At present, the target platform for the cross-compiler is the .NET platform. It is planned to develop a code generator that translates the IR3 representation (see below) into the MSIL code, which is the assembly language of the .NET virtual machine. At the moment, the Cloud-Sisal-compiler uses the IR3 internal representation translator in the C# program as the code generator, which is then translated into the MSIL code by the C# compiler. A library has been developed that contains a C# class system that provides support for the execution period for Cloud-Sisal-programs. In addition, the user can use these classes to provide interaction between the Cloud Sisal program and the C# code (for example, to organize I / O).

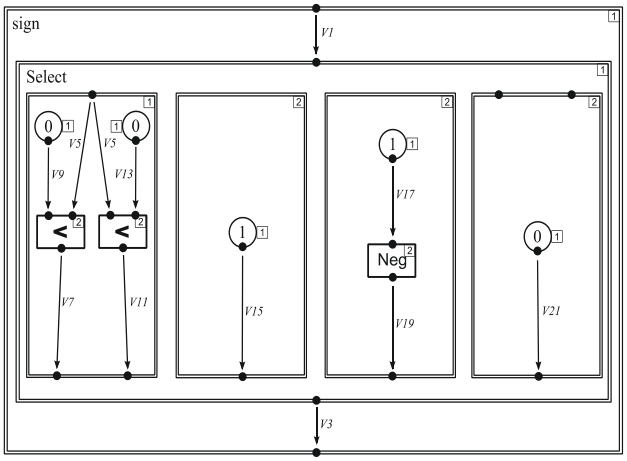

Fig. 2. The R2 presentation of the sign funtion.

4 Conclusion

The project of the CSS system for supporting of functional and parallel programming is 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 on high performance parallel computers without extensive rewriting and debugging.

The CSS system can open a real world of supercomputers for a wide range of applied programmers without requiring large investments in new computer systems and allowing to unload existing supercomputers from their inefficient use for designing and debugging parallel programs.

Its application not only can make supercomputers included in the network available to a wide range of application programmers, but also can increase the reliability of the parallel programs being created, since it allows us to formulate solutions of problems on an abstract level in a declarative style and without binding to specific

computing resources, to check some correctness properties of annotated functional programs by using formal methods and to build efficient parallel code by using compilers. Application of the technology also improves the efficiency of supercomputers by transferring the work of programmers in designing and debugging programs from expensive supercomputers to cheaper and more conventional personal computers, and by eliminating the need for a programmer to construct, verify and debug a program for solving the same problem each time anew when transferring the program from one supercomputer to another is needed.

The main segments of the market for the application of the developed methods and system are scientific application, the solution of national economic problems and training of personnel. Among the potential users of the system are a large team of scientists, designers and engineers who are involved in solving applied problems that impose increased demands on information and computing resources, as well as students and teachers studying parallel programming. The existing tendencies to increase the number of applied programmers involved in solving increasingly complex problems, and to the development of telecommunications networks and supercomputing centers, not only do not reduce, but substantially expand, the requirements for programming systems of this type, creating an expanding market for their mass application.

At present, the CSS system consists of experimental versions of web interface, interpreter, graphic visualization/debugging subsystem, optimizing cross-compiler and cluster runtime. The current target platform for the Cloud-Sisal-compiler is .NET. The compiler 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.

The work was partially supported by the Russian Science Foundation (grant 18-11-00118).

References:

[1] J. Backus. Can programming be liberated from the von Neumann style? *Commun. ACM*, Vol.21, No.8, 1978, pp. 613–641.

[2] J. McGraw, S. Skedzielewski, S. Allan, et all. *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.

[3] A.P. Ershov. The Transformational Machine: theme and Variations, *Lecture Notes in Computer Science*, 1981, Vol. 118, pp. 16-32.

[4] V.N. Kasyanov, E.V. Kasyanova. Graph- and cloud-based tools for computer science education, *Lecture Notes in Computer Science*, Vol. 9395, 2015, pp. 41 - 54.

[5] D.C. Cann. Retire Fortran?: a debate rekindled, *Commun. ACM*, Vol. 34, No. 8, 1992, pp. 81–89.

[6] J.-L. Gaudiot, T. DeBoni, J. Feo, et all. The Sisal project: real world functional programming, *Lecture Notes in Computer Science*, Vol.1808, 2001, pp. 45–72.

[7] J.T.Feo, P.J. Piller, S.K. Skedzielewski, et all. SISAL 90. In: *Proceedings of High Performance Functional Computing*, Denver, 1995, pp. 35–47,

[8] V.N. Kasyanov. Sisal 3.2: functional language for scientific parallel programming, *Enterprise Information Systems*, Vol. 7, No. 2, 2013, pp. 227-236.

[9] D.C. Cann, J.T. Feo, A.P.W. Böhm, et all: *Sisal Reference Manual: Language Version 2.0*. Tech. Rep. Lawrence Livermore National Laboratory, UCRL-MA-109098, Livermore, CA, 1991.

[10] V.N. Kasyanov. Methods and tools for structural information visualization, *WSEAS Transactions on Computers*, Vol. 12, No. 7, 2013, pp. 349–359.

[11] U. Brandes, M. Eiglsperger, J. Lerner, and C. Pich. Graph Markup Language (GraphML), In: *Handbook of Graph Drawing and Visualization*. CRC Press, 2013, pp. 517–541.

[12] V.N. Kasyanov, T.A. Zolotuhin. Visual Graph – a system for visualization of big size complex structural information on the base of graph models, *Scientific Visualization*, Vol. 7, No. 4, 2015, pp. 44 – 59. (In Russian).

[13] V.N. Kasyanov. Transformational approach to program concretization, *Theoretical Computer Science*, Vol. 90, No. 1, 1991, pp. 37-46.

[14] V.N. Kasyanov. A support tool for annotated program manipulation, In: *Proc. of Fifth European Conf. on Software Maintenance and Reengineering*, IEEE Computer Society Press, 2001, pp. 85–94.