A Visual Cloud System for Parallel and Functional Programming Teaching and Learning

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

Abstract: - In this paper, a visual cloud system being under development for supporting of functional and parallel programming teaching and learning is considered. The input language of the system is a functional language Cloud Sisal that exposes implicit parallelism through data dependence and guarantees determinate result as well as supports data types and operators typical for scientific calculations such as loops and arrays. The system is aimed to provide means to write and debug Cloud-Sisal-programs on low-cost devices as well as to translate and execute them in clouds.

Key-Words: - Computer science education; functional programming; hierarchical graph representations; parallel programming; visual cloud system

1 Introduction

Academic research and engineering challenge both require high performance computing, which can be achieved through parallel programming. The existing curricula of most universities do not properly address the major transition from single-core to multi-core systems and from sequential to parallel programming. As a rule, they focus on applying of application program interface (API) libraries and open multiprocessing (OpenMP), message passing interface (MPI), and compute unified device architecture (CUDA)/GPU techniques. This approach is useful but misses the goal of developing students' long-term ability to solve real-life problems by "thinking in parallel".

We can see that the history of computing has shown shifts from explicit to implicit programming. In the early days, computers were programmed in assembly language, mostly with the purpose of utilizing the available memory space as effectively as possible. This came at the cost of obscure, machine-dependent, hard to maintain programs, which were designed with high programming efforts. High-level languages were introduced to make programming more implicit, portable and less machine-dependent. With the advent of massively parallel computers and their promise of hundreds of gigaflops, we have seen a return to the explicit programming paradigm. Using these languages with explicit message passing library routines as "machine languages", people attempt to utilize the available processing power to the largest extent, again at the cost of high programming effort, machine-dependent, and hard to maintain code. A compiler for an implicitly parallel programming language alleviates the programmer from the task of partitioning program and data over the massively parallel machine.

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.

In the paper, the system CSS (Cloud Sisal System) being under development at the Institute of Informatics Systems is considered. It is aimed to be an interactive visual environment for supporting of functional and parallel programming teaching and learning. 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 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.

2 The CCS 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 intended to provide a general-purpose user interface for a wide range of parallel processing platforms (See Fig. 1). In our conception, the cloud interface gives transparent ability to execute programs on arbitrary environments. The JavaScript client does not demand installation; small educational 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.

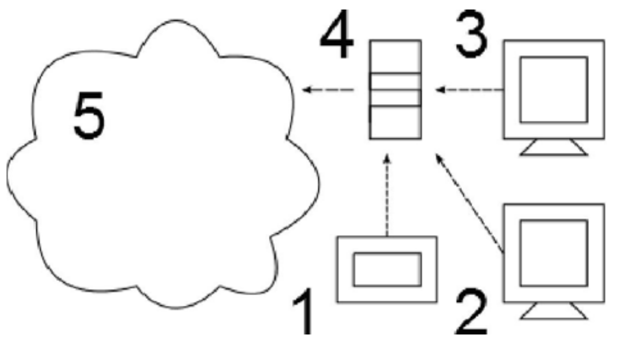

Fig. 1. Cloud service structure: 1, 2 and 3 – clients, 4 – cloud access server, 5 – execution environment.

The CSS system uses a functional language Cloud Sisal as its input language and a language of so-called hierarchical graphs [2] as the internal representations of Cloud-Sisal-programs.

The CSS system includes five main parts: web interface, interpreter, graphic visualization / debugging subsystem, optimizing cross-compiler, cluster runtime. The interpreter is available on web via a browser; it translates a source Cloud-Sisal-program to its hierarchical graph representation (so-called the first internal representation of the source Cloud-Sisal-program) 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. Web interface contains also some usual parts like syntax highlighting, persistent storage for program code, authorization and so on.

3 Cloud Sisal Language

Functional language Sisal (Steams and Iterations in a Single Assignment Language) is considered as an alternative to FORTRAN language for supercomputers [3, 4]. 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, 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 [5] for many supercomputers (e. g., SGI, Sequent, Encore Multimax, Cray X-MP, Cray 2, etc).

The Sisal 90 language definition [6] 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 [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. 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.

Consider, for example, a Cloud Sisal program for matrix multiplication (Fig. 2).

The first two statements define type names for arrays. Note that no sizes are provided, and all Cloud Sisal aggregate data instances are dynamically created, resized, and de-allocated at runtime. Only the dimensionality and element types are relevant to the type specifications.

The header for the Mult function shows that two TwoDim arguments and three integer arguments are expected, and one unnamed value will be returned. The returned value is two dimensional array of double precision reals, but again, only typing and not sizing is specified. Name can be bound to this returned value at the site of invocation of the function if the programmer wishes. An invocation of a function is semantically equivalent to the reproduction of the function code at that site, with appropriate argument substitution. This equivalence, called “referential transparency” is a fundamental property of functional languages, and is responsible for the strengths of the Cloud Sisal language. This strength lies in a simplified analysis process for the compiler. Two functions can run in parallel if no data dependency exists between the functions. The same function with equivalent inputs will always returns equivalent values.

```
type OneDim = array[ double_real ];
type TwoDim = array[ OneDim ];
function Mult (A,B:TwoDim;
               K,N,M:integer
               returns TwoDim)
   for I in 1, K cross J in 1, M
      S := for L in 1, N
              R := A[I,L]*B[L,J]
              returns value of sum R
           end for
      returns array of S
   end for
end function
```

Fig. 2. Cloud Sisal program for matrix multiplication.

All Cloud Sisal expressions, including whole functions and programs, evaluate to value sets. In the example (Fig. 2), the Mult function evaluates one array, which is the value of the for-expression contained in the function definition. This for-expression is a loop construct, which is an indicator of potential parallelism to the Cloud Sisal compiler. This loop has an index range defined as the cross product of two simpler ranges. This means that the body of the loop will be instantiated as many times as there are values in the index range, in this case *K*M*, and each body instantiation will be independent, if no data dependencies exist among them. It should be noted that the set of independent loop bodies can be executed in parallel or not, based on the compiler's and the runtime system's analyses of their costs, as well as on options and annotations specified by the programmer. Reductions are used to determine returning values of loops. Keyword “returns” at the end of a loop is followed by the name of a reduction and its parameters. For example, in the Mult function the reduction of the inner loop is used to summarize the all values of *R* and the reduction of the outer loop is used to make an array from all values of *S*.

4 Single Assigment

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 [9]. 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.

5 Loops and Arrays

The Cloud Sisal language 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 *i*-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
```

In functional programming every statement is a function returning the value, the 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 use following construction of the 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 a pure functional language, 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[1]. 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. 3).

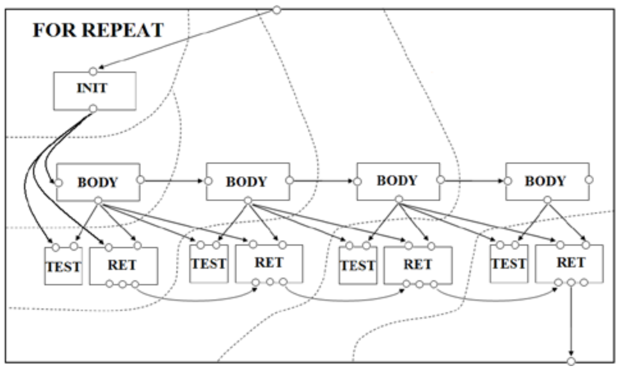

Fig. 3. Post-conditional (for repeat) pipelined structure.

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. Arbitrary subarray selection is provided beyond the rectangular subsets available in some other notations. Many infix operations operate element-by-element on array operands and a useful set of functions on arrays is defined. A subarray update facility allows safe alteration of array values. Many applications are expressible succinctly with these features. Array generation, selection and update may use vector subscripts to refer to arbitrary, non geometric sections of arrays.

1 More information can be found at http://threadingbuildingblocks.org/

6 Annotated Programming

The Cloud Sisal language supports also so-called annotated programming and concretizing transformations [9, 10] and includes so-called pragma statements in the form of formalized comments (optimizing 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. 4. Cloud-Sisal-program with optimizing annotation.

According to the approach used [9, 10], any source program is considered as a base for constructions of a number of different specialized programs. Every construction starts with the annotated general-purpose program which consists of 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 annotated program transformations is performed (either automatically or interactively with the user), which results in a specialized program being correct and more qualitative for this specific context of application.

An example of optimizing annotations is an assert pragma statement. Every expression in 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 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 usage of the assert pragma statements please consider factorial function declaration and definition which are represented in Fig. 4.

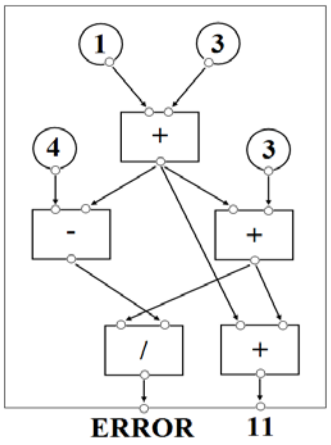

Fig. 5. Error value propagation in "always finished computations" semantics

Another example of optimizing annotations is a pragma "parallel" which can be used before a case expression in Cloud 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.

7 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 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. Let us consider the following example Java-program:

```
try {
   for (int i=0; i<N; i++) {
      a[i]=a[i]/((i+1)%K);
    }
} catch (Exception e) {
   //display 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. 5).

8 Internal Representations

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

IR1 is a language of hierarchical graphs [2] 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.

IR2 is an extension of IR1 but is not applicative. It introduces operations that explicitly allocate and manipulate memory and also introduces a new class of operations, which are similar to IR1 nodes except that they are told where in memory to construct their results. Also, artificial dependence edges are added to define additional synchronization constraints where they may be useful. Finally, data edges can be decorated with pragmas to specify access rights to the data they transmit and to allow operations to modify their inputs.

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

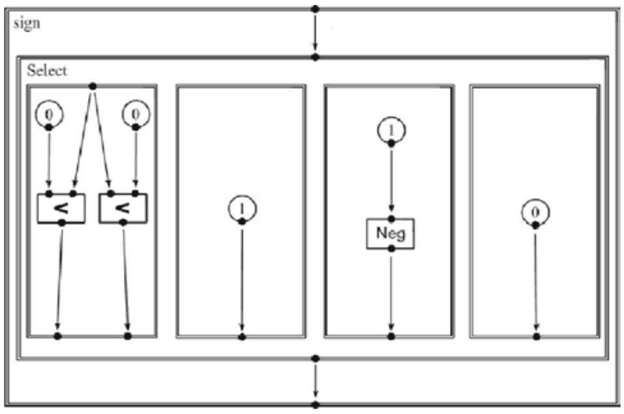

Fig. 6. A function sign and its IR1-representation.

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.

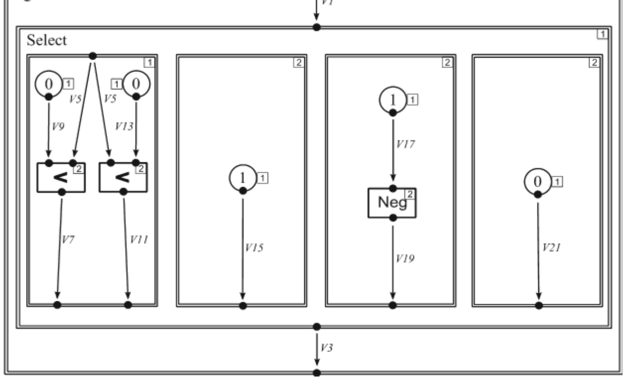

Fig. 7. IR2-representation of the function sign.

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;
   }
```

9 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.

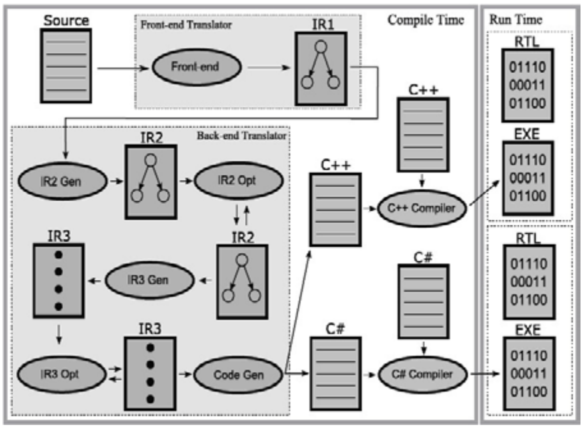

Fig. 8. The Cloud-Sisal-compiler and run-time support.

After parallelization, CodeGen generates C++ or C# code, and the compilation can be completed using the target machine's C++ or C# compiler.

The optimizing cross-compiler generates also a GraphML-file with a graph which represents data structures handled by the compiler. GraphML (or Graph Markup Language [12]) 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 [13]. 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.

10 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.

The Pifagor language is currently developed at Siberian Federal Institute [14]. This language is optimized to dataflow graph description; syntax is not easy to understand because it differs from common imperative and functional languages. For example, it has no infix operations, no loops. The following Pifagor function performs vector multiplication by scalar:

```
VecScalMult << funcdef Param
// Argument format: ((x1, x2, : xn),
y),
// where ((x1, x2, : xn) is a vector,
y – 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 it is aimed on the list processing and the conception of unlimited parallelism scheduled as limited at runtime.

This project has compiler and interpreter used for scientific proposes: development of the new scheduling algorithms and parallel programming education.

The F# language [15] 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.

At one hand: F# is functional ML-family language; functional paradigm suits better for parallel computations. At the other: it 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; it is hybrid, you can write implicitly parallel and sequential programs both. Multithreaded programming on F# is quite similar to C# or C programming.

Not in case of the only F# but for the all functional languages developers are trying to make language programming available for wide range of people 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 makes the semantic sequential or non-deterministic.

11 Conclusion

The project of the CSS system for supporting of functional and parallel programming teaching and learning 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 the world of parallel and functional programming to all students and scientists without requiring a large investment in new, top-end computer systems. A smaller number of high speed computers can be shared among all scientists because parallel development is moved to low-end systems.

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.

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.

The work was partially supported by the Russian Foundation for Basic Research (grant 15-07-02029).

References:

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

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

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

[4] 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.

[5] 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.

[6] 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,

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

[8] 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.

[9] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck: Efficiently computing static single assignment form. *Transactions on Programming Languages and Systems*, Vol. 13, No. 4, 1991, pp. 451-490.

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

[11] 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.

[12] 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.

[13] 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).

[14] L. Legalov: Functional language for creation of architecture-independent parallel programs. *Computational Technologies*, Vol. 10, No. 1, 2005, pp. 71-89. (In Russian).

[15] D. Syme, A. Granicz, A. Cisternino: *Expert F#3.0.* Apress, 2012.