Support tools for functional programming distance learning and teaching

V N Kasyanov, E V Kasyanova and A A Malishev
A.P. Ershov Institute of Informatics Systems SB RAS, Ac. Lavrentieva ave., 6,
Novosibirsk, 630090, Russia
E-mail: kvn@iis.nsk.su

Abstract. The cloud parallel programming system CPPS being under development at the A.P. Ershov Institute of Informatics Systems SB RAS is based on the Cloud Sisal language which carries on the traditions of previous versions of the Sisal language while remaining a functional data-flow language focused on writing large scientific programs and expanding their capabilities by supporting cloud computing. In this paper, the online environment of the CPPS system which allows a user on any device with Internet access to develop and execute functional programs in the Cloud Sisal language is considered.

1. Introduction

Functional programming is a programming paradigm being entirely different from the conventional imperative 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 first functional programming language was Lisp, developed in 1961 by the American scientist J. McCarthy. Although the language has gained wide popularity due to its greater expressiveness and elegance compared to traditional languages, its applicability has been limited mainly to tasks of artificial intelligence. The new era of functional programming began with the 1978 Turing lecture by the inventor of Fortran J. Backus [1]. This new understanding and wider acceptance of functional programming was determined primarily by the process begun in these years to move to considering the programming problem in its full context, starting with the specification of the problem and the logical analysis of its solvability, the by-product of which is the program itself. The emergence of computing systems with parallel architectures further increased the importance of functional programming, since it allows the user to free from most of the complexities of parallel programming inherent in imperative languages, and entrust the compiler with the issues of building a program that is efficiently executed on a computing system of a particular parallel architecture. In addition, many technical problems in systems and application programming are clarified when solutions are presented in a functional style.

The cloud parallel programming system CPPS being under development at the A.P. Ershov Institute of Informatics Systems SB RAS uses so-called the Cloud Sisal language as its input functional language [6, 7]. It is assumed that within the framework of the CPPS system a programmer will be able in clouds through a browser to create, debug and verify a Cloud Sisal program on his low-cost device in a visual style not taking into account the target supercomputers. Then he can use the CPPS system to tune the developed functional program to a supercomputer available him via network in
order to achieve high performance execution of the constructed parallel program, as well as to transfer this constructed program to the supercomputer to run it and receive its results.

The Cloud Sisal language continues the tradition of previous versions of the Sisal language [2, 5] and remains a functional data-flow language focused on writing large scientific programs, but expands their capabilities for supporting cloud computing. Its functional semantics guarantees deterministic results for parallel and sequential implementations – something that cannot be guaranteed for usual imperative languages like C or Fortran. Moreover, the implicit parallelism of the Cloud Sisal language removes the need to rewrite the source code when transferring it from one computer to another. It is guaranteed that any Cloud Sisal program correctly executed on a personal computer will be correctly executed on any high-speed parallel or distributed computer.

The paper considers the online environment of the CPPS system which allows a user on any device with Internet access to develop and execute functional programs in the Cloud Sisal language. The rest of the paper is organized as follows: Section 2 outlines the Cloud Sisal language, Section 3 describes the online environment of the CPPS system, Section 4 presents the profiler of the CPPS system, and Section 5 concludes the paper and highlights the future research directions.

2. The Cloud Sisal language

The Cloud Sisal language has the usual advantages of functional programming languages, such as, for example, pure functions or reference transparency (that is, each function call is semantically equivalent to the reproduction of the function code at the call site with the corresponding change of parameters) and single assignment (that is, each variable in a program is defined only once), but contains arrays and loops that are not inherent in functional languages, uses the semantics of so-called “always completed calculations”, and supports so-called annotated programming.

Consider the following fragment of the Cloud Sisal program with arrays and loops:

type Matrix = array [.., ..] of integer;

function Mult ( A, B : Matrix; M,N,L : integer returns Matrix )
    for i in 1, M cross j in 1, L do
        R:= for k in 1, N do returns sum of A[i, k] * B[k, j] end for
        returns array [.., ..] of R
    end for
end function

The first line defines \textit{Matrix} as the type name for the array type. Only the form and type of elements is specified in this specification of array type. In particular, the dimensions are not specified in it, and all instances of the described composite data type must be dynamically created, changed, and deleted during program execution.

The second line is the header of the \textit{Mult} function which indicates that two five arguments (two \textit{Matrix} arguments: \textit{A} and \textit{B}, and three integer argument: \textit{M}, \textit{N} and \textit{L}) are expected, and one \textit{Matrix} result as unnamed value is calculated (returned). So, for a returned array of a function only the type of its elements and its shape can be indicated, not its size. A name can be bound to this returned array at the place where the \textit{Mult} function is called if the programmer needs it.

Any expression in Cloud Sisal programs, including any function entirely, computes a set of values. In particular, the \textit{Mult} function computes one two-dimensional array of integers, which is the value of the loop expression contained in the body of its definition. This loop has an index range which is defined as the Cartesian product of two simpler ranges for \textit{i} and \textit{j}. This means that the body of the loop will be executed as many times as there are values in the range of indices, in this case \textit{M} * \textit{L}, and all instances of the body will be independent, since there are no data dependencies between them. So, this set of independent instances of the body can be executed in parallel, and concrete kind of their parallel execution can be selected on the basis of both the costs associated with the compiler and the target computing system and the options set by the programmer.
The name $R$ inside the loop body should not be considered as reusing the name in the sense of assigning a variable in an imperative program. Here, this name is used to denote value in the loop body, and in fact it most likely will not actually exist in the executable program.

All $M \times L$ instances of the loop body for different values of $i$ and $j$ will independently calculate specific instances of integer values which are defined as the sum of $A[i, k] \times B[k, j]$ for all $k$ from 1 to $N$. All these individual values will be collected together in a couple of array and returned as the result of the loop. The positions of the values in the result array, as well as the dimensions of the returned array, are determined by ranges of cycle indices. In this case, a two-dimensional array with an index from 1 to $M$ in the first dimension and an index from 1 to $L$ in the second dimension will be returned.

The Cloud Sisal language offers the user a rich set of various standard reductions for loops (such as sum and array reductions considered above), and also allows to define new reductions. The use of reductions in loops is good in that their implementation may depend on the target computing system. When a Cloud Sisal program is executed in a single-threaded environment, all reductions will be performed sequentially, but when it is executed in several threads some reductions can be executed in parallel.

The Cloud Sisal language uses the semantics of so-called “always completed calculations”, which means that the execution flow of any Cloud Sisal program never stops and always returns the resulting values even if any erroneous situation occurs. For this, the set of values for each data type (both predefined and user-defined) is augmented by one error value. This value is produced whenever an operation cannot return a legal value. For example, a Boolean type consists of the tree values: true (true), false (false) and error value (error[Boolean]). It is assumed that unless otherwise stated, and some arguments of operations on built-in types (or predefined functions) are erroneous, their results will be erroneous values as well. It is always possible also to find out if the value of an expression is wrong, using a special operation. This property of the Cloud Sisal language can help in analyzing and optimizing the Cloud Sisal programs, as well as in parallel programming in the Cloud Sisal language, in particular in facilitating assisting try-catch mechanism for error handling.

The Cloud Sisal language supports so-called annotated programming [3, 4], and it allows the user to describe the semantic properties of the source (or basic) program, which are known to him, in the form of annotations being formalized comments. Any comment that begins with the dollar symbol “$” is called an annotation (or a pragma) and sets the properties of the basic program construction that follows (one construction can be supplied by a few annotations). Annotations can be used by compiler (and other subsystems of the CPPS system) to improve efficiency and reliability of the constructed parallel program.

An annotation can have the form “name” or “name = list of expressions”, where basic program names that are visible at the location of the annotation can take part in list expressions. As a rule an annotation is placed before a program expression and describes its property. The result of a unary expression in the annotation to which it refers is denoted by a single underscore “_”, and the results of an $n$-ary ($n > 1$) expression is denoted as “_ [1]”, ..., “_ [n]”. An annotation can be placed also in function declarations either before the returns keyword or in front of the first formal parameter, and describes either a property of the returned values which should be valid just after executing function body or property of the parameters which should be valid just before executing function body.

Let us give some examples of annotations.

Before each expression an annotation “assert = Boolean condition” can be placed. The Boolean condition of the annotation describes assertion, which should be true immediately after the expression is evaluated. It is also allowed to use in an annotation-assertion so-called extended Boolean condition, which has either the form “(all <names> : <Boolean condition> : <extended Boolean condition>)” or the form “(is <names> : <Boolean condition> : <extended Boolean condition>)” and defines the scope for the name specified in it. For example, the extended Boolean condition (all $i : i > 2 : A \ [i] = 0$) is true if all elements in the array or stream $A$ are zero for which the index is greater than two, and the extended Boolean condition (is $i : i > 2 : A \ [i] = 0$) is true if there is at least one zero element in $A$ with an index greater than two.
For example, the annotation-assertion in the header of the following function definition

```plaintext
function Mult ( //$ assert = M=N; N=L; (all i, j: i!=j: A[i, j]=0 & B[i, j]=0) A, B : Matrix; M,N,L : integer returns Matrix )
  for i in 1, M cross j in 1, L do
    R:= for k in 1, N do returns sum of A[i, k] * B[k, j] end for
  end for
end function
```

sets that the function will be always used for such matrices that both matrices are square and diagonal, and therefore the function can be equivalently converted to the following one:

```plaintext
function Mult ( //$ assert = M=N; N=L; (all i, j: i!=j: A[i, j]=0 & B[i, j]=0) A, B : Matrix; M,N,L : integer returns Matrix )
  for i in 1, M cross j in 1, M do
    R:= if i=j then A[i, i] * B[j, j] else 0 end if
  end for
end function
```

An annotation “non_used = list of values” before an expression sets that the all listed values of the expression becomes unnecessary immediately after its calculation (that is, they are not used in the future when the program is executed) and can be removed from the program. For example, the nonused-annotation in the header of the following function definition

```plaintext
function generate ( N : integer /*$ non_used = [...2] */ returns Matrix, Matrix)
  for i in 1, N cross j in 1, N do
    returns array [..., ...] of i * j; array [..., ...] of i + j
  end for
end function
```
sets that the second result of the function will be never used and its calculation in the function body can be deleted as follows:

```
function generate ( N : integer /* non_used = _[2] */ returns Matrix, Matrix )
  for i in 1, N cross j in 1, N do
    returns array [..., ..] of i * j
  end for
end function.
```

3. The online environment

The current version of the CPPS system uses the developed online environment [12], which supports user interaction with the existing components of the CPPS system via a web browser. In particular, the existing environment allows a registered user to develop programs in the Cloud Sisal language, run them and receive results in a web application.

![Figure 2. The Cloud Sisal program.](image)

The user can store several projects, and each project can consist of several modules. Both a project and a module can be created by clicking the appropriate button and entering a name and, optionally, a description. Each project and each module can be described in the corresponding fields in the system editor both before and after its creation (see figure 1).

It is proposed to describe the input data of the program (arguments of the main function) in JSON format. For example, a user can describe input data as following:

```
{
  "M": 15, // number
  "a": 15.1, // number
  "V": [1, 2, 3, 4.0], // vector
  "A": [
    [1, 2, 3],
    [4, 5, 6]
  ] // matrix
}
```

where the vector V and the matrix A are actually just numeric arrays of different dimensions.

To simplify understanding the format and setting the input data by a user, the system can generate for the user on the base of the main function description the text of an example of setting its input data. To obtain this, the user must first click the “Generate from main function” button above the data entry field. After this the user can simply enter his values into the automatically generated template for the input data of his main function.
The online environment of the CPPS system is created using the django framework and is built on the principles of the service architecture. The services use LXC for Linux systems and VirtualBox for Windows, and systems exchange data (for example, for compilation of the source program) through HTTP requests. At present, two services are implemented: the editor service (which also performs both construction and controlled execution of the C# code) and the compiler service on Windows. In a Windows service, a server written in Python is responsible for receiving and processing requests.

```csharp
// This file is generated by Sisal 3.1 compiler.

using System;
using System.Threading;
namespace Sisal_C_sharp
{
    public class SISAL_CODE
    {
        public static bool Ftb(CI32 V_1, out CI32 V_3)
        {
            CI32 V_7;
            CI32 V_17;
            CI32 V_21;
            CI32 V_27;
            CI32 V_31;
            V_7 = (new CI32(V_1 < new CI32(0x2)));
            if (V_7.Value == true)
            {
                V_3 = V_1;
            }
            else
            {
                V_17 = V_1 - new CI32(0x1);
                V_27 = V_1 - new CI32(0x2);
                Ftb(V_17, out V_21);
                Ftb(V_27, out V_31);
                V_3 = V_21 + V_31;
            }
            return true;
        }
        public static bool main(CI32 V_33, out CI32 V_37)
        {
            Ftb(V_33, out V_37);
            return true;
        }
    }
} // namespace Sisal_C_sharp
```

**Figure 3.** C#-code produced by the compiler.

To edit the code in the Cloud Sisal language in the front-end part, the ACE editor [11] with syntax highlighting is used, which is built into the Web application. The jQuery UI library [14] was used to create dialog boxes and controls. To simplify the development of the front-end part, the jQuery library was also used [13]. The last two libraries, despite the reputation of being outdated, perfectly cope with emerging tasks and, due to their lightness, make the editor more accessible to work on a wide range of user devices.
4. The profiler

The current version of the CPPS system contains the Sisal 3.1 compiler, which converts modules written in the Cloud Sisal language into classes written in C#, which can be used to compose programs in C# and compiled them by the system with using MONO or .NET.

Consider, for example, a C# code shown in figure 3. This code is obtained by the Sisal 3.1 compiler from the program shown in figure 2. It is a C# class containing images of functions in the Cloud Sisal language, and this code is included in the MONO project. The system generates a MONO solution using the built code and data with variables in JSON format. It is required to generate the code of the main class, which includes the initialization of the input data and the call of functions from the classes generated by the compiler. The input data is used to generate the main solution class and is placed there as hardcoded data. The Sisal 3.1 compiler was developed using COM technology. For this reason, the system uses a separate (virtual) Windows server for the compiler.

If the source code of the module has not changed since the last broadcast, then the resulting code saved in the database is used for it, so as not to waste time on unnecessary repeated (sometimes very time-consuming) compilation.

MONO-project is executed under the control of the system. At the same time, the execution time is limited to take into account those cases when the program cannot complete due to infinite recursion (or an infinite loop) or takes too long for its execution or profiling.

The results of the program's work, its profiling information and description of all errors which occur during compilation and execution are transmitted to the frontend by the server side and shown to the user in a separate window.

![Figure 4. Live view of the profiling result.](image)
One of the tasks in the implementation of the system is the profiling of programs (figure 4). This process allows the user to better understand his program and explore its bottlenecks, that is, those segments of it, the processing of which takes up the greatest amount of system resources, mainly processor calculations. The process of profiling functional programs has a number of peculiarities, for example, the sequence of calculations of individual functions is not known in advance for it [10].

To profile a Cloud Sisal program, the C# code obtained during its compilation is supplied with code inserts in the form of auxiliary functions and structure (see figure 5). These functions are placed at the beginning and at the end of function bodies. They are aimed to store the start and end time of the function execution in microseconds, the threads in which the function was executed, the values of the arguments, the functions passed, as well as the nesting of calls (which function was called by which function). After the program instrumented in this way completes its work, the collected information is used to prepare an interactive report on the program's work.

In the report, the collected data is visualized as an interactive call tree in SVG vector format. The svgwrite Python library is used to generate the image. A graphical interactive report (figure 4) is a web application segment that displays a tree of function calls on a plane in the form of rectangles that represent calls and contain information about how long it took to execute this call and what arguments it was executed with. Subtrees that are too large for rendering are displayed as separate vertices, but the user can always expand any subtree by clicking on the image of the corresponding vertex. The horizontal offset and width of the vertex image correspond to the time (start and duration) of the corresponding function call. The call image also contains the identifier of the called function, the values of the arguments and the duration of its execution.

Figure 5. Automatic code insertion for profiling a functional program.

5. Conclusion
The paper describes the created online environment of the cloud-based parallel programming system CPPS, which, together with the created compiler and system profiler, allows teaching functional
programming to the user if he has a device with Internet access. Its use can occur as follows. First, the registered user creates the texts of the functional program and its input data in the editor. Then the program text is translated into C# on the translator server, the input data is also translated into C# and, together with the source code, is used to form MONO-Solution, which is then compiled and executed. The results of execution are data in the stdout file as well as a JSON file with profiling data, which are then used to build a final report presented to the user.

The use of other created tools of the CPPS system allows users of the online environment presented in the paper through a web browser to visually debug and formally verify functional programs in the Cloud Sisal language, as well as to execute parallel programs automatically built by the compiler according to their functional specifications [6, 8, 9].

Acknowledgements
The authors are thankful to all colleagues taking part in the project described.

References
[1] Backus J 1978 Can programming be liberated from the von Neumann style? Functional style and its algebra of programs Commun. ACM 21(8) 613–641
[2] Gaudiot J-L, DeBoni T, Feo J, et al 2001 The Sisal project: real world functional programming Lecture Notes in Computer Science 1808 pp 45–72
[3] Kasyanov V N 1989 Program annotation and transformation Programming and Computer Software 15(4) 155-164
[4] Kasyanov V N 1991 Transformational approach to program concretization Theoretical Computer Science 90(1) 37-46
[5] Kasyanov V 2013 Sisal 3.2: functional language for scientific parallel programming Enterprise Information Systems 7(2) 227–236
[6] Kasyanov V N, Gordeev D S, et al 2020 The cloud parallel programming system CPPS: visualization and verification of Cloud Sisal programs (Novosibirsk: NSU) (In Russian).
[7] Kasyanov V N and Kasyanova E V 2019 Methods and system for cloud parallel programming Proceedings of the 21st International Conference on Enterprise Information Systems (ICEIS 2019) SCITEPRESS 1 pp 623–629
[8] Kasyanov V N, Kasyanova E V and Kondratyev D A 2020 Formal verification of Cloud Sisal programs J. Phys.: Conf. Ser 1603 012020
[9] Kasyanov V N, Zolotuhin T A and Gordeev D S 2019 Visualization methods and algorithms for graph representation of functional programs Programming and Computer Software 45(4) 156–162
[10] Sansom P M and Jones S P L 1993 Profiling lazy functional programs Proceedings of the 1992 Glasgow Workshop on Functional Programming pp 227–239
[11] The Ace editor, https://ace.c9.io/
[12] The CPPS system, http://cpps.iis.nsk.su
[13] The jQuery, http://jquery.com
[14] The jQuery UI, http://jqueryui.com