Methods and System for Cloud Parallel Programming

Victor N. Kasyanov and Elena V. Kasyanova
Institute of Informatics Systems, Lavrentiev pr.6, Novosibirsk, 630090, Russia

Keywords: Annotated Programming. Graph Presentation of Functional Programs. Functional Programming. Parallel Programming. Programming Language.

Abstract: In this paper, a cloud parallel programming system CSSP being under development at the Institute of Informatics Systems is considered. The system is aimed to be an interactive visual environment of functional and parallel programming for supporting of computer science teaching and learning. The system will support the development, verification and debugging of architecture-independent parallel programs and their correct conversion into efficient code of parallel computing systems for its execution in clouds. In the paper, the CPPS system itself, its input functional language, and its internal graph presentation of the functional programs are described.

1 INTRODUCTION

Parallel computing is one of the main paradigms of modern programming, but the existing curricula of most universities do not properly address the major transition from single-core to multi-core systems and sequential to parallel programming. They focus on applying application program interface (API) libraries and open multiprocessing (OpenMP), message passing interface (MPI), and compute unified device architecture (CUDA)/GPU techniques. This approach misses the goal of developing students' long-term ability to solve real-life problems by “thinking in parallel”.

Functional programming 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 first language of functional programming was Lisp, developed in 1961 by the American scientist J. 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 inventor Fortran J. Beckus “Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs” (Backus, 1978). 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 be free 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.

The development of functional methods of parallel programming successfully continued in the late 70s in the languages of VAL and BARS, as well as in a number of more modern projects, such as DCF, Pythagoras, COLAMO, and the SISAL language (an abbreviation with the English expression Streams and Iterations in a Single Assignment Language) (Gaudiot et al., 1995), the first version of which refers to 1983. SISAL is developed as a functional programming language, specifically oriented to parallel processing and the replacement of the Fortran language on supercomputers in scientific computing.

It is still early to speak about real displacement, but SISAL as a parallel programming language is quite interesting itself and has already found its application in dozens of organizations around the world. There are several implementations of the SISAL language (version 1.2 (McGraw et al., 1985)) for supercomputers, in particular Denelcor HEP, Vax 11-780, Cray-1, Cray-X / MP, TERA, * T, TAM and MIDC. The Livermore National Laboratory and Manchester University have developed an improved version of the SISAL language (SISAL-90 (Feo et al., 1995)), which has not yet been implemented.

In this paper, the cloud parallel programming system (CPPS) being under development in the Program construction and optimization laboratory of the Institute of Informatics Systems with support of the grant of the Russian Science Foundation (project 18-11-00118) is considered. Main properties of the CPPS system itself, its input functional language (Cloud Sisal language) and its internal graph representation of Cloud Sisal programs are described.

2 CLOUD PARALLEL PROGRAMMING SYSTEM CPPS

Modern approaches to the development of parallel programs are mostly architecturally oriented, when the created programs to achieve effective work are closely related to the parallel computing architecture on which they are executed and, as a rule, are developed. Therefore, the requirements for the qualification of developers of parallel programs are very high, especially since testing and debugging a parallel program is much more complicated than a sequential one, and the problem of verifying parallel programs is very far from solving not only practically but also theoretically. At that, only a little part of domestic users has access to high-performance computing equipment, which in terms of the number of supercomputers and their total capacity is quite inferior to those available in developed countries and is concentrated in a relatively small number of places outside which parallel programming is not developed, but the main part of applied programmers works.

Moreover, in modern computer technology there is a constant change of architectural paradigms, which, in turn, leads to the problem of portability of already developed parallel programs. We have to constantly adapt the already created product to the changed hardware. This is due to the fact that different parallel computing systems have their own resource limitations, which must be taken into account during the development of the program. Carrying out such adaptations is a very intellectual task, requiring substantial rewriting of parallel programs and performing again their verification and debugging. As a result, adapted parallel programs often contain new errors and are not as effective as they should and could be.

Therefore, it seems very promising to carry out a project to develop language and software tools that support the construction, verification and debugging of architecture-independent parallel programs as well as the correct conversion them into efficient code for parallel computing systems of various architectures using semantic transformations.

Methods will be developed and an experimental version of the cloud extensible integrated visual parallel programming system CPPS will be developed. The input language of the CPPS system is the Cloud Sisal language (Kasyanov, Kasyanova, 2018) which continues the tradition of previous versions of SISAL (such as Sisal 90 (Feo et al., 1995) and Sisal 3.2 (Kasyanov, 2013)) while remaining a functional data-flow language focused on writing large scientific programs and expanding their capabilities by supporting cloud computing. The functional semantics of Cloud Sisal guarantees deterministic results for parallel and sequential implementation — something that cannot be guaranteed for traditional imperative languages like Fortran or C. Moreover, the implicit parallelism of the language removes the need to rewrite the source code when transferring it from one computer to another. It is guaranteed that the Cloud-Sisal-program, correctly executed on a personal computer, will be guaranteed to be correctly executed on any high-speed parallel or distributed computer.

Wherein, annotated programming methods and concretizing transformations used in the CPPS system will allow us within the framework of the declarative programming style to adapt the portable parallel programs to the task classes and the architecture of the supercomputer, while preserving their correctness, and also to obtain a more efficient parallel code by using during adaptation knowledge of the user about the task, program and computer, expressed in annotations.

CPPS is developed as an integrated cloud programming environment in the Cloud Sisal language, which contains both an interpreter that supports interactive user interaction when creating and debugging a functional program, and an optimizing cross-compiler that builds a parallel program according to its functional specification.

Using the CPPS system, an application programmer will be able to develop, verify and debug a Cloud-Sisal-program in a visual style and without taking into account the target supercomputer, and then use the optimizing cross-compiler to tune the debugged program to one or another supercomputer available to him in network, in order to achieve high performance execution of the parallel program, as well as transfer the built program to the supercomputer to run it and receive its results.

The CPPS system uses an internal graph representation of functional programs that is focused on their visual processing and is based on attributed hierarchical graphs (Kasyanov, 2013). It supports the construction of visual images of graph internal representations of Cloud-Sisal-programs and their use in the construction of correct functional programs (Kasyanov, Kasyanova, Zolotuhin, 2018). It is assumed that the system will also support the construction of visual representations of the internal data structures that arise in the cross-compiler when parallel programs are constructed as well as of the dynamic processes that occur when the parallel programs are executed. These visual representations can help users to control optimizations during cross-compilation to improve efficiency of the compiled parallel programs.

3 CLOUD SISAL LANGUAGE

The Cloud Sisal language has the usual advantages of functional programming languages, such as, for example, 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.

Consider the following fragment of the Cloud Sisal program:

```
type OneDim = array [..] of integer;
type TwoDim = array of OneDim;
function generate
   ( N : integer
   returns TwoDim, OneDim
   )
   for i in 1, N cross j in 1, N
   do
     A := i * j;
         B := i + j
 returns array [.., ..] of A;
     array of B
 end for
end function
```

The first two lines define the type names for the arrays. It can be seen that the dimensions are not specified in them, and all instances of the described composite data types must be dynamically created, changed, and deleted during program execution. Only the form and types of elements are contained in these specifications of array types. In the second line (in the TwoDim type definition), the form is omitted and by default it is assumed to be [..].

The header of the “generate” function indicates that one integer argument, “N”, is expected, and two unnamed values are calculated (returned). Each return value is an array of integers, but again, only the shapes of the arrays are indicated, not their sizes. Names can be bound to these returned values at the place where the function is called if the programmer needs it.

A function call is semantically equivalent to the reproduction of a function code at the call site with the corresponding change of parameters. This equivalence, often referred to as “reference transparency”, is a fundamental property of functional languages and is one of the strengths of the Cloud Sisal language. This property in particular simplifies the analysis processes performed by the optimizing compiler, since the functions have no side effects and are deterministic. In other words, any two functions can be executed in parallel, if there is no data dependency between functions, and the same function with the same actual parameters always returns the same values. 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 N * N, and all instances of the body will be independent, since there are no data dependencies between them. Those sets of independent cycle bodies that will be executed in parallel, and which will not, will be selected on the basis of the costs associated with the compiler and the system, and also on the options set by the programmer.

All Cloud Sisal program expressions, including functions entirely, compute sets of values. In the above case, the generate function computes two-dimensional and one-dimensional arrays, which are the values of the expression contained in the function definition. The specified expression is a loop construct that tells the compiler for Cloud Sisal about potential concurrency. This cycle has an index range, defined as the Cartesian product of two simpler ranges. 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 N * N, and all instances of the body will be independent, since there are no data dependencies between them. Those sets of independent cycle bodies that will be executed in parallel, and which will not, will be selected on the basis of the costs associated with the compiler and the system, and also on the options set by the programmer.

The names "A" and "B" inside the loop body should not be viewed as reusing these names in the sense of assigning a variable in an imperative program. Here, these names are used to denote values in the loop body, and in fact they most likely will not actually exist in the executable program. The important point here is that each instance of the loop body, containing specific values for i and j, will independently calculate specific instances of integer values, defined as i * j and i + j; then all of these individual values will be collected together in a couple of arrays and returned. The positions of the values in the result arrays, as well as the total size and dimension of the returned arrays, are determined by their shapes and ranges of cycle indices. In this case, two arrays are returned, each of which consists of N^2 integers: a two-dimensional array with an index from 1 to N in each dimension and a one-dimensional array with an index that varies from 1 to N^2. The use of temporary names in the loop is optional, and the above return condition can be rewritten as follows:

```
returns array [.., ..] of i * j;
        array of i + j
```

without changing the final results. With such a change, the body of the loop will become empty, and essentially the language treats the expressions in the "array of" as anonymous temporal.

The language offers the user a rich set of various standard reductions, and also allows the definition and use of its own reductions. The use of reductions is good in that their implementation may depend on the target computing system. When a program is executed in a single-threaded environment, the reduction can be performed sequentially, but when executed in several threads it can be executed in parallel.

The Try-catch mechanism is quite popular today for error handling, but this approach has conflicts with parallel program execution. When an exception occurs, all execution threads must be stopped, the pipeline is cleared, etc. Also, there are difficulties with maintaining software determinism in the case of parallel execution and the occurrence of exceptions. For the Cloud Sisal language, such problems do not exist, because it uses the semantics of "always completed calculations", which means that the Cloud Sisal program execution flow never stops and always returns the resulting value (possibly containing "error" values) even if any erroneous situations. For this, there is a distinguished erroneous value in each type, for example, a Boolean type consists of the values of true (true), false (false) and error value (error [Boolean]). Unless otherwise stated, and any arguments of operations on built-in types or predefined functions are erroneous, their results will also be erroneous values. It is always possible to find out if the value of an expression is wrong, using a special operation.

The language supports annotated programming (Kasyanov, 1989) and concretizing transformations (Kasyanov, 1991), allowing the user to describe the semantic properties of the program, which are known to him, in the form of formalized comments. A comment that begins with the dollar symbol "$" is called an annotation (or a pragma) and sets the properties of the construction that follows (one construction can be compared with several annotations). The result of a unary expression in the annotation to which it refers is denoted by a single underscore "_", and the arity of an n-ary (n> 1) expression is denoted as "_ [1]", ..., "_ [n]". An annotation can have the form "name" or "name = list of expressions", where names that are visible at the location of the annotation can take part in list expressions. Unrecognized annotations cause compiler warnings.

Let us give some examples of annotations.

Before each expression there can be an annotation "assert = Boolean condition", which should be true immediately after the expression is evaluated.

The assertions can be placed in function declarations both before the returns keyword and impose conditions on the returned values, and in front of the first formal parameter and set conditions on the formal parameter names specified in them, which should be valid when the function is called immediately before executing body function.

It is allowed also to replace the Boolean condition in the assertion with the so-called extended Boolean condition, which has either the form "(all <name>: <Boolean condition>: <extended Boolean condition>)" or the form "(is <name>: <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 condition (is *i*: *i*> 2 : *A* [*i*] = 0) true if there is at least one zero element in *A* with an index greater than two.

For example, the assertion in the header of the function definition indicates that the specified function is always used for exponentiation when exponent is a power of two:

```
function power
(
//$ (is k : k>=0 & k<n : n = 2**k)
x : real n : integer returns real
)
 if n =0 then returns 1
 elseif n%2 =0 then
     returns (power(x, n/2)**2
 else   returns (power(x, n-1))*x
 endif
end function
```

and therefore it can be equivalently converted to the following function:

```
function power
(
//$ (is k : k>=0 & k<n : n = 2**k)
x : real
n : integer
returns real
)
 if n =0 then  returns 1
 else returns (power(x,n/2)**2
 endif
end function
```

Before each expression there may be a "non_used = list of values" annotation, which indicates the values becomes unnecessary immediately after the calculation of the expression (they are not used in the future when the program is executed) and can be removed from the program. For example, as indicated in the annotation below the second result of the "generate" function is never used

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

and its calculation in the function body can be deleted:

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

4 INTERNAL REPRESENTATIONS OF CLOUD SISAL PROGRAMS

The CPPS system uses an internal graph representation (IR) of Cloud Sisal programs, which is focused on their semantic and visual processing and is based on the attributed hierarchical graphs (Kasyanov, Kasyanova, 2013). It is assumed that the IR representations of the Cloud Sisal programs are shown to users of the system along with their textual representations and are used by users for the purpose of visual debugging of the Cloud Sisal programs and their controlled optimization (Kasyanov, Kasyanova, Zolotuhin, 2018). It is assumed also that the Cloud Sisal program is assembled from IR-modules (both in the interpreter and the compiler) before interpreting it or optimizing translation.

In developing the internal presentation, the following essential requirements were taken into account.

1. Machine independence for both representation of parallelism (there is no explicit splitting of computations into several streams), and for the values (independence from the capacity of the machine architecture) data types.
2. Completeness of the internal representation, allowing to translate any design of the source language into a semantically equivalent fragment of the internal representation.
3. The possibility of relaying into a syntactically correct program after the transformations of the internal representation of the program, preserving its semantics.
4. Simplicity of interpretation (execution given by the internal representation of calculations) without any additional transformations of the internal representation.
5. Structuredness of objects of internal representation for the task of the natural nesting of some constructions of the original programming language into others.
6. All implicit actions on data, such as type conversions, must be explicitly expressed using objects of internal representation.

7. Extensibility, in the sense of easily introducing new objects of internal representation to define new programming language constructs and data types.

There are several ways of defining an internal representation, for example, in this capacity we can consider the parse tree of the program being broadcast. However, in our case, it does not satisfy the requirement of interpretability, since this requirement implies the availability of contextual (semantic) information. At the same time, the requirement of machine-independent parallelism and the functionality of the represented programming languages lead us to use data flow graphs as a natural basis for the structure of the internal representation. Thus, the IR graph, in contrast to the control flow graph (CFG), commonly used in optimizing compilers for imperative languages (such as C or Fortran languages), expresses not the control flow, but the data flow in the program. Data flow graphs have several useful properties for the required internal representation, including the following.

1. Explicitly specified information (semantic) links (arcs) between operand operations (vertex ports) make the interpretation process feasible without additional transformations. This implies the absence of side effects of computations (due to the absence of the concept of a variable) — the natural property of purely functional languages.

2. Parallelism at the level of individual informational independent operations, independent of the machine architecture.

The vertices of the IR graph correspond to the program expressions, and the arcs reflect data transmissions between the vertex ports, the ordered sets of which are assigned to the vertices as their arguments (input ports or inputs) and results (output ports or outputs). By virtue of the property of the Cloud Sisal language, this graph is acyclic and does not contain two arcs entering the same input.

Vertices denote operations on their inputs (arguments), the results of which are at the outputs of the vertices. There is, however, a special kind of vertices denoting literals (constants) of any type, each of which has one output and an empty set of inputs.Vertices are simple and composite. Simple vertices (or simply vertices) have no internal structure and represent elementary operations, such as, for example, plus or minus. Composite vertices (or fragments) correspond to the composite expressions of the Cloud Sisal program, such as, for example, a loop expression or function body, and additionally directly contain sets of vertices corresponding to the expressions they consist of. For each fragment *F*, the number and types of the set of vertices *M* that are directly contained in it, as well as the set of arcs (*p*, *q*) that exist between its ports and the ports of these vertices, are determined by the type (or semantics) of the compound expression, but satisfy the following property: p is the output of some vertex from *M* or the input of the fragment *F*, and q is the input of some vertex from *M* or the output of the fragment *F*. It is clear that the fragment *F* in addition to the above vertices of *M* and the arcs incident to their ports, which are directly contained in *F*, will contain other vertices and arcs of the graph if there are fragments among the vertices of *M*, but due to the properties of the Cloud Sisal language there are no arcs among them which are incident to the ports of fragment *F* and are not directly nested in *F*.

5 CONCLUSIONS

The CPPS system is intended to provide means to write and debug parallel programs regardless target architectures on low-cost devices and then execute them in clouds on high performance parallel computers without extensive rewriting and debugging.

So, it 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. Using the CPPS system, any user will be able to develop, verify and debug a Cloud-Sisal-program in a visual style and without taking into account the target supercomputer, and then use the optimizing cross-compiler to tune the debugged program to one or another supercomputer available to him in network, in order to achieve high performance execution of the parallel program, as well as transfer the built program to the supercomputer to run it and receive its results.

The CPPS system can be used also for teaching and learning of optimizing compilation and high performance computing.

The use of the CPPS system can also increase the efficiency of using supercomputers by transferring the work of programmers to design and debug programs from expensive supercomputers to cheaper and more familiar personal computers, as well as by eliminating the need for a programmer to build, verify and debug a program to solve the same problem each time when switching from one supercomputer to another.

ACKNOWLEDGEMENTS

The authors are thankful to all colleagues taking part in the project described. This work was carried out with a grant from the Russian Science Foundation (project 18-11-00118).

REFERENCES

Backus, J., 1978. Can programming be liberated from the von Neumann style? Commun. ACM, **21** (8), 613–641.

Feo, J. T., Miller, P. J., Skedzielewski, S. K., Denton, S. M., Solomon, C. J., 1995. SISAL 90. In: Proceedings of High Performance Functional Computing. pp. 35-47, Denver.

Gaudiot, J.-L., DeBoni, T., Feo, J., et al., 2001. The Sisal project: real world functional programming. In: Pande, S., Agrawal, D.P. (Eds.) Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems. LNCS, vol.1808, pp. 45-72, Springer, Heidelberg.

Kasyanov, V. N., 1989. Program annotation and transformation. Programming and Computer Software, **15** (4), 155-164.

Kasyanov, V. N., 2013. Sisal 3.2: functional language for scientific parallel programming. Enterprise Information Systems, **7** (2), 227-236.

Kasyanov, V. N., 1991. Transformational approach to program concretization. Theoretical Computer Science **90** (1), 37-46.

Kasyanov, V. N. Kasyanova, E. V., 2013. Information visualization based on graph models. Enterprise Information Systems, **7** (2), 187-197.

Kasyanov, V. N., Kasyanova, E. V., 2018. Programming Language Cloud Sisal. Preprint IIS 181, Institute of Informatics Systems, Novosibirsk. (in Russian)

Kasyanov, V. N., Kasyanova, E. V., Zolotuhin, T. A., 2018. Visualization of graph presentations of data-flow programs. WSEAS Transactions on Information Science and Applications, **15,** 140-147.

McGraw, J., Skedzielewski, S., Allan, S., Grit, D., Oldehoeft, R., Glauert, J., Dobes, I., and Hohensee, P., 1985. 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.