Sisal: parallel language development

Idrisov Renat
Program construction and optimization laboratory
IIS Sib RAS
Novosibirsk, Russia

In this paper we present unfinished research on Sisal language currently held at our institute. It wasn't our development initially but now we are trying to create better science computational task solving with it. I also describe why this research is valuable and its current state.

Language description, functional languages, dataflow languages, parallelism

I. Introduction

Parallel computations are more than actual today, the architecture of the popular computing systems is changed almost every year, a developer needs some universal method of describing a parallel algorithm independent to the system. The idea is to make algorithm description closer to the task description and not to implement the exact algorithm but to formulate the problem as far as it possible. Of course, this idea can be found at early A. P. Ershov articles and is not new in general, but it become more relevant today. Functional programming can give the developer an ability to write programs like problem statements and it is better for exploration of parallelism. In this article we briefly describe some of Sisal programming language constructions and its benefits for parallel algorithm forming.

II. Advantages

A. *Single assignment*

Sisal [1] [2] differs from other functional languages and we think that this difference make Sisal more adapted for computational tasks. First of all, it has some usual functional language benefits like single assignment[3]. 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.

B. *Streams and arrays*

Sisal also uses arrays and loops which is not common for a functional language, but it is good for computation: you don't have to worry about the recognition of the tail recursion or the number of iterations or matrix description which is simpler with arrays.

You can operate with n-th element of the array in a natural way like in Fortran:

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

C. *Verbose syntax*

And the last benefit is more verbose syntax. It makes program source more readable and as the result – long time development by different people becomes easier. Many functional languages suffers from the lack of the words in the program source, it makes the text hard to understand. The example below is the famous Haskell1 quicksort:

```
qsort []     = []

qsort (x:xs) =

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

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

```
function qsort (Data : array[real] returns array[real] )

if array_size( Data ) > 2 then

let

L, Middle, R := for E in Data

 returns array of E when E < Data[ 1 ]

array of E when E = Data[ 1 ]

array of E when E > Data[ 1 ]

 end for

in
```

[1] More information on Haskell can be found at http://www.haskell.org

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

III. Loops and reductions

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. Sisal is pure functional, it has no side effects and any loop contains the reduction call, also user can implement his own reductions.

The reductions are good because its implementation can depend on target system. When the program is executed in single-threaded environment it can be performed sequentially, but when executed on multiple threads it can be performed in parallel. Similar idea can be found in modern library "Threading Building Blocks" by Intel[2]. This library allows usage of reduction mechanism in C++, but user can also use ordinary loops as well. In Sisal programs reductions can't be avoided.

In Sisal we have three kinds of loops: Post-conditional, pre-conditional and "for all" (operation is applied to the 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. 1”.

At this figure loops are divided into parts: Initialization, loop body, loop test, loop reduction (ret) and range generator, we think that the part names can briefly describe them, but if you need more information – please check Sisal language description [1] [2].

Using reductions matrix multiplication can be implemented meaningfully:

[2] More information can be found at http://threadingbuildingblocks.org/

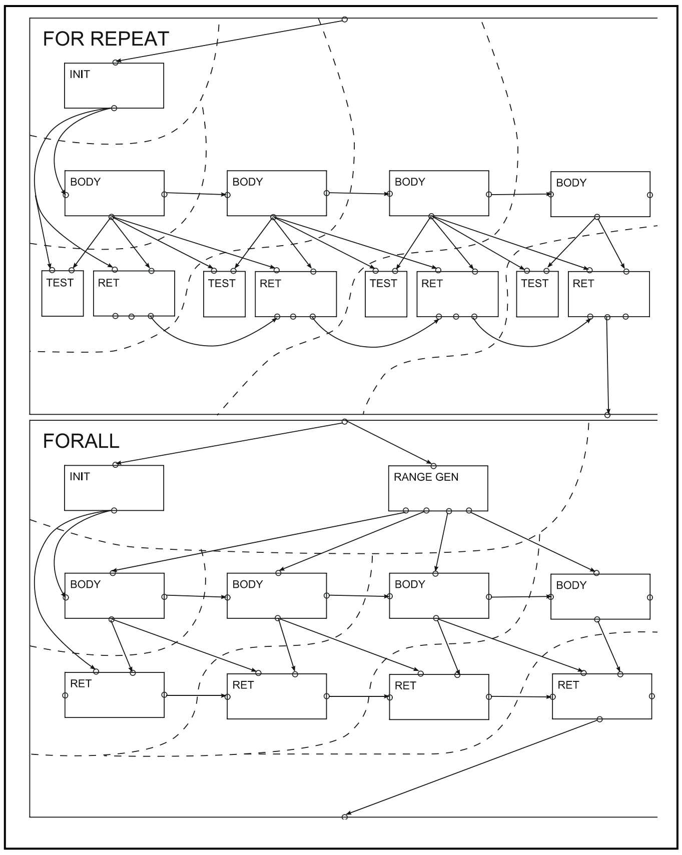

Figure 1. “for all” and post-conditional (for repeat) pipelined structure

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

Reduction can be always used in sequential style:

```
function multiply(A:array[array[real]]; B:array[array[real]];
N:integer
                returns array[array[real]])
  for i in 1, N cross j in 1, N
  returns array of
    for initial
      s := 0.0;
      k := 1
```

```
        while (k <= N)
        repeat
          s := old s + A[i, old k] * B[old k, j];
          k := old k + 1
        returns value of s
        end for
      end for
    end function
```

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

IV. 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. Check the following JAVA example:

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

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

In 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. 2”).

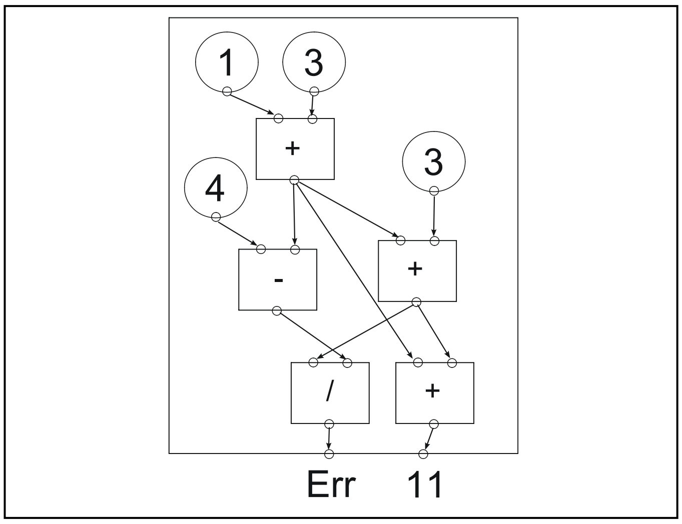

Figure 2. Error value propagation in “always finished computations” semantics

V. 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.

A. *Pifagor*

This language is currently developed at Siberian Federal Institute [4]. The 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.

B. *F# from Microsoft*

We can’t say that F# 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.

VI. Our current state

At our institute we develop both language standard and compiler. First version of the language was derived from Sisal 1.2 initially developed at Lawrence Livermore National Laboratory [5]; current version of the language is 3.2 and we are trying to improve it by solving some science computation problems.

Sisal language compiler is used mostly by its developers for scientific proposes: developing new optimization and analysis algorithms, checking and improving language standard.

The main aim for today is to make language available for people who solving computational problems and students. We are developing JavaScript Sisal interpreter to achieve these objectives.

My personal contribution is algorithm implementation for language standard improvement investigations; backend optimization algorithms and parallelization at Sisal compiler. Now I develop JavaScript Sisal interpreter.

VII. Conclusion

The main idea of this paper is to explain some algorithmic solutions and language properties valuable for parallel execution. Development of the new languages become more and more easy, user can create his own science field specific language. And if the programs became task definitions and not algorithm descriptions – it will be not necessary to rewrite it when the execution environment or computation system will change. You have to rewrite only the compiler; it is easier than re-solve all the tasks again. Even if the language will be specific it will give advantages while migrating to another executor. Microsoft pays additional attention to functional programming and provides tools for functional language building, it is positive trend3.

Sisal language was initially developed for parallel programming and writing Sisal programs is not the same as writing C programs. In this paper we briefly described semantic difference. If you are interested in parallel programming and unfamiliar with functional or logic paradigm you should definitely explore it.

References

[1] V. Kasyanov, A. Stasenko, “Sisal 3.2 language structures decomposition”, Lecture Notes in Electrical Engineering. — Berlin: Springer-Verlag, 2009. — Vol. 28. — P. 582–594.

[2] J. McGraw, S. Skedzielewski, S. Allan, D. Grit and R. Oldehoeft “Sisal: Streams and iterations in a single assignment language, Language Reference Manual Version 1.1”, Lawrence Livermore Nat. Lab. Manual M-146. — Livermore, CA 1983.

[3] R. Cytron, J. Ferrante, B. Rosen, M. Wegman and K. Zadeck, “Efficiently computing static single assignment form,” Proc. POPL-1989, pp. 25-35, ACM.

[4] A. Legalov “Functional language for creation of architecture-independent parallel programs” Легалов А. И. Функциональный язык для создания архитектурно-независимых параллельных программ Вычислительные технологии : журнал. — 2005. — Т. 10. — № 1. — С. 71-89

[5] McGraw, J. R. et. al. “Sisal: Streams and iterations in a single assignment language, Language Reference Manual Version 1.2” Lawrence Livermore Nat. Lab. Manual M-146 (Rev.1). — Livermore, CA 1985.

[3] This tools are provided with F# can be found at http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/