A performance analysis of the R language and an assessment of the capabilities for its improvement

Lidija Fodor*, Srđan Škrbić*
* University of Novi Sad/Faculty of Science/Department of Mathematics and Informatics, Novi Sad, Serbia
lidija.fodor@dmi.uns.ac.rs, srdjan.skrbic@dmi.uns.ac.rs

Abstract— **R is considered both as a programming language and an environment for statistical computing. It provides a wide variety of functionalities and almost limitless opportunities to extend. Its easiness to use, in combination with the interpreted nature, ensures its popularity in different areas of business and science. However, when it comes to more demanding computations in terms of performance, R turns to be significantly slower, compared to other languages. As the need for data analysis increases rapidly, the existence of a simple and at the same time efficient tool is of inestimable importance. This paper strives to present an analysis of performance drawbacks of R, and exposes an idea for possible improvements.**

I. INTRODUCTION

R is an interpreted language, used for data analysis in various fields. The language was designed in 1993 as a successor to S, by Ross Ihaka and Robert Gentleman [7]. In 1995, R was released under a GNU license. The language covers a wide range of functionalities, grouped into packages. CRAN [17] and Bioconductor [18] are well-known, commonly used repositories of R packages.

The use of R spreads through different areas of science including bioinformatics, mathematics, data-mining. On the other hand, it also finds its application among others in finance, telecommunications, pharmaceuticals, commonly used as a tool for data-mining, graphs construction and forecasting. Many of the world-wide known companies base their business decisions on the results of analysis, conducted by R.

The performance of R is partly influenced by the constructions used in the program, but they are also the results of some design decisions behind the language. We will discuss the best and most efficient ways of writing programs in R, but the focus will be on performance issues of the language itself.

In this paper, we will make the following contributions:

- Performing an analysis of R's performance, through concrete, real life examples, with comparison to other languages.
- Indicating how these problems could be solved, using modern concepts of building new language implementations.

Our goal is to compare the capabilities of R to other languages and tools as Java and MATLAB, based on practically usable examples. This approach reflects the main advantages of R, related to its conciseness and easiness to use. These are the features that established the place of R in the world of data analysis. We will show that in some cases, the performance of R can be much better, with regard to mentioned compared languages. Still, when it comes to large amount of data being analyzed or the use of complicated algorithms that require nested loops semantics, the performance issues become evident.

The reasons for inefficient program execution can be relatively easily defined ,and we will try to identify them in the fourth section of this paper. However, the solution that offers the same advantages as R, without significant syntax changes, that ensures fast programs, is still an open question that many developers are interested in.

II. RELATED WORK

A tendency to increase the execution speed of R programs arises naturally. Hence, several authors proposed some innovative ideas, about performance improvement in R.

Morandat et al. made a detailed evaluation of the design of the language [1]. They made a formal report of the semantics of the core of R and also introduced the TraceR framework for analysis, with an implementation and language evaluation review.

An interesting initiative is the work of Eddelbuettel and Sanderson [3]. They use the C ++ language with extensions in form of packages, called Rcpp, relying on Armadillo C ++ library, in order to work with matrices. In addition, they apply a template-based framework for metaprogramming, with the aim to easily convert R's linear algebra algorithms to C++.

Li et al. introduce the principle of automatic and transparent parallelization, using a runtime framework called pR [5]. The idea of semi-automatic parallelization also occurs in the work of Jiang et al. [6]. This approach introduces the idea of adding pragma directives to the source code in OpenMP style.

An equally interesting approach is program specialization at the level of the interpreter, developed by Wang et al. [4]. The idea relies on a direct optimization of the virtual machine using R's extensions ORBIT VM, in order to avoid allocation, by the principle of aggressive deletion of allocated objects and shortening the instructions paths.

FastR is a project, developed by Kalibera et al., which represents a unique subset of the implementation of the R language, built within the host language Java [2]. This approach uses the principle of an AST interpreter, and is based on ANTLR parser generator, which creates a tree, on the basis of the source code. The execution is achieved by an executable tree, while the conversion between the trees happens during an evaluation process.

We propose an idea for solving the performance issues of the language by developing a completely new

implementation, without changing its syntax. However, the underlying implementation should provide optimizations and possibly parallelization, in such a way that the user should not be aware of them.

III. KEY FEATURES OF R

R is a functional, object-oriented and dynamic language. This unique combination of features makes the language flexible and plain to the end users. Its interpreted nature, combined with the wide range of existing functions, makes it possible to perform various complex computations with a single function call.

A. Functional nature

In accordance with its functional features, R treats functions as priority entities, which makes it possible to assign them to variables or pass them as arguments to another functions. From the aspect of scopes, R uses lexical scoping, so the range of each value can be determined statically, before run-time. The global scope, called "environment", can contain implicitly created sub scopes for the need of loops and functions, but explicit user-defined sub scopes are also allowed. In addition, dynamic binding allows the insertion of names to scopes dynamically.

Function arguments are evaluated lazily, by packing them into so called "promises", containing the argument's expression and the information about the current scope. That way, promises are evaluated as they become needed.

R provides high-level flexibility, when declaring function arguments. First of all, default values are enabled. Besides that, a variable number of parameters can be also specified, treated as an arbitrary size array. When calling functions, the parameters can be passed positionally, named or combined, with the restrictions that each call should contain at least one positional parameter, as well as that after a variable list of arguments, the named approach should be used.

These items provide a convenient tool for the end users to work with functions, without deep understanding of how they actually work.

B. Dynamic nature

Dynamic language features are manifested through dynamic evaluation and dynamic type system. The dynamic type system frees the source code from explicit type declarations. As a consequence, the same variable can hold values of different types at different places in the code. Assigning a type to a value, as well as conversions between them are performed implicitly. R supports numeric, logical and character as primitive types.

Dynamic evaluation is supported through the function "eval", and the reverse process of converting an evaluated expression to its string form is also present. As mentioned before, R enables reflection over environment, by means of direct access and modification from the program. This creates an excellent base for flexible program manipulation and extensions creation.

C. Object-oriented nature

R supports two different object models, called S3 and S4. S3 is the older approach and is also called "single-dispatch generic function system". It is not an object-oriented system in the true sense, as it does not introduce classic object-oriented concepts. Instead, it relies on setting class attributes for variables. It should be mentioned, that variables can have different attribute values as their integral parts, which describe some features of the variable, like size or type. The class attribute only serves to define methods over the class. When a method is called for a variable with a defined class attribute, the method defined for that class will be executed.

The S4 object model is also called "classes and multi-methods system". This is a newer approach, with all the standard object-oriented features. It allows declaring classes with "slots" for holding values, methods, as well as defining hierarchies of inheritance.

Although the S3 system has greater popularity due to its simple nature and longer history, the S4 system brings the real object-oriented note to the language, and opens further possibilities for program creation.

D. Basic elements of R programs

The basic data type in R is the vector type. Each primitive value is treated as a single-element vector. Simple binary operators are applicable to vectors, with the semantics of applying the operator element-wise. The length of one vector should be a multiple of the length of the another, but the operator can be applied even if this condition is not satisfied. The main feature of vector type is that all its elements have to be of the same type. R also supports the value NA (Not Available), which is important in statistical computations. Matrices, lists, data-frames, environment and functions are considered also as non-primitive data types.

While matrices have the usual meaning, lists should be explained more carefully. In R, they represent heterogeneous vectors, which mean that they can hold different number of vectors of different sizes and different types. This makes them extremely flexible for representing heterogeneous data. Data-frames are similar to lists, but restricted to vectors of the same size.

R supports common control structures, as the if branching and iterations in forms of for, while and repeat loops. As an alternative to loops, R introduces a family of "apply" functions. These functions can be used for different data structures, by applying some arbitrary function on the structure. Apply functions are the preferred way for iterating over data-structures, as they result with better performance than the direct use of loops.

To perform a piece of job, R commands (assignments and calls), can be grouped inside scripts, or can be directly entered to R's command line one by one. Extending the present possibilities of the language can be easily achieved by creating the desired functions and putting them to packages.

The R community mainly includes end users that interactively calls existing functions for reading a piece of data, performing some analysis and graphically represent the results. From this aspect, the clearness and simplicity of the language are the most important demands.

There are two more groups of R users: experts from the field of statistics and programmers. The wide range of statistical algorithms is available thanks to statisticians. On the other hand, programmers are the group with deeper understanding of the features of the language.

The question is whether it is sufficient to write as much as possible optimal functions at background, so that they could perform efficiently when the users at the front end use them. The answer is negative. A large number of R's functions is written with care, to the extent that they are implemented in C programming language, in order to gain as much as possible speedup. Despite that, they do not show significant execution time improvements. The reasons for this and some other performance problems will be described below.

IV. PERFORMANCE

A. Basic notes

Despite the described advantages of R, working with large data sets shows serious deficiency in performance. Earlier studies emphasized some of the weakest points of the language. During their detailed language design analysis, Morandat et al. evaluated performance related problems [1].

Firstly, a significant amount of time is spent on memory management. The reasons for this are linked to the basic features of the language: allocating space for vectors, copying arguments, garbage collection, built-in functions calls. Copying arguments is the result of passing by value, so regardless of the size of the structure that represents the argument, a copy is created. Also, all the functions that work with values defined in an upper environment, have their own local copies of the values. A certain amount of time is spent on lookup and pairing values with arguments, hence the language is interpreted with dynamic binding.

An interesting fact is that a large number of provided R functions, is written in another languages, usually C and Fortran, due to performance issues. While these functions contribute to a more efficient execution, the amount of time spent calling them is greater than it might be expected. On average, a fifth of execution time is spent on calling such functions.

Memory consumption is another important aspect that influences efficiency. Beside the process of allocation for user defined data, a significant amount of memory is allocated internally for the interpreter. Allocating a vector for a simple value leads to unnecessary occupation of memory and to time consumption for deallocation later. Allocation and deallocation happen on the heap, which further affects performance. These are small components of the total execution time and may seem insignificant, but their combination and multiple application can seriously impact performance.

The dynamic nature of the language directly reduces the possibilities for optimizations. The advantages in terms of simplicity and clarity certainly have a cost in terms of speed. The combination of a loop and dynamic type system can lead to inefficient running of the program.

```
x<-5
a<-c()
for(i in 1:10000){
  a[i]<-i*x
}
print(a)
```

Listing 1 – Illustration of the dynamic typing problem.

Consider the small example, given in Listing 1. As the variable x is not declared with type, each iteration of the for loop has to decide on which data type should it apply the multiplication operator. This is a trivial case, but thinking of more complicated situations, which require working with methods in object-oriented concepts, illustrates the problem even more.

While declaring functions, one should carefully decide what is the minimal number of arguments needed, as more arguments can lead to slower execution. The reason for this is the lazy evaluation, which packs the arguments into promises. Practically, the moment of the evaluation of promises often happens immediately, so the costs of creating the promises affect the program. Even with the simplest functions that just apply a binary operation, each additional argument increases the execution time by a few milliseconds.

The costs of calling functions can be also an issue. The interpreter needs to create a scope for the function call, copy the values and add the arguments to the scope. The fact that arguments can be named, and that variable parameter lists could be used, requires additional effort.

B. The preffered ways of writing programs

Although the nature of R is the main reason for performance issues, it is worth to mention that the right way of writing programs inside it can decrease the execution time to certain degree.

When operating on vectors, it is important to keep in mind that loops perform inefficiently. Vectorized functions can, on the other hand, operate on vector elements, with the efficiency, as with primitive values. Let's consider the example of creating a vector, containing prime numbers in range of 1 and 100000.

Listing 2 shows the source code using a for loop, while Listing 3 illustrates the use of vectorized function. On a Quad core AMD Phenom 9550 machine, with 2.2 GHz and GNU/Linux(x86-64), used for all examples here, the execution time of the first version of the program is 0.8 seconds. Running the second approach gives an execution time of 0.12 seconds. It can be noticed that the vectorized approach is nearly 6 times faster. For more complicated uses, the difference could be even larger.

Another important note is that recursion should be avoided, hence R does not support tail recursive calls optimizations. Also, anonymous functions should be used instead of named ones, where possible.

```
library(gmp)
v<-1:100000
w<-c()
for(i in seq(along=v)){
  if(isprime(v[i]))
    w[length(w)+1]<-v[i]
}
```

Listing 2 – Prime numbers, loop-based approach.

```
library(gmp)
v<-1:100000
w<-v[isprime(v)!=0]
```

Listing 3 – Prime numbers, vectorized approach.

Respecting these rules, the real performance of R programs can be observed, as deficiency caused by bad style of programming is eliminated.

C. Case-studies

As R is often used for data-mining, consider its application when working with time-series data. Listing 4 shows the use of hierarchical clustering on time-series data, loaded from a file, applying the similarity measure DTW and plotting the resulting dendrogram. The library DTW contains the function for applying dynamic time warping (dtw) similarity measure. The test file contains 100 time series, each 5000 points long. The values of points are between 0 and 1.

Reading the data is achieved by simply calling the function read.table, while the distance matrix is created by calling dist. The results are recorded in a csv file. The function hclust is used to perform hierarchical clustering, and the results are used to create a dendrogram. The code is self-descriptive and very easy to understand. The whole process of reading the data, performing clustering, and writing the results into a file is achieved by several function calls.

It is clear, that the program is written respecting the previously described recommendations for writing the most efficient code. The code is mainly consisting of predefined function calls. The execution time of this program is 18.19 seconds. This is quite a large number for a dimension of 5000 points of time-series. Practically, it is often necessary to work with much larger data series. To identify time consumption rates of different parts of the program, we will try to remove particular calls and compare the timings. If the dendrogram creation is eliminated, the execution time decreases to 6.26 seconds, which indicates how costly the plotting can be.

```
start<-proc.time()
library(dtw)
name<-"TestData.csv"
fName<-substr(name,0,nchar(name)-4)
data<-read.table(name, header=F, sep=";",
                 row.names=1)
distMatrix<-dist(data, metod="DTW")
write.table(as.matrix(distMatrix),
            paste(fName, "DTW", ".csv",
                  sep=""))
hc<-hclust(distMatrix, method="average")
jpeg(paste(fName,"DTWavg",".jpg", sep=""))
plot(hc, main=paste(substr(fileName, 0,
                    nchar(fileName)-4),
                    "DTW average",
                    "linkage", sep=" "))
dev.off()
end<-proc.time()
elapsed<-end-start
print(elapsed)
```

Listing 4 – Case study, time-series data-mining.

Secondly, if the reading from a file is replaced by creating a matrix filled with random data, the time will be 0.95 seconds. The conclusion is that data input/output represents the main problem in this kind of application. Data-mining always require a lot of communication with files, so the time spent for that is usually not negligible.

However, comparing the same time-series data mining program, written in Java and MATLAB, one can make the conclusion, that R is the best choice, regarding to the amount of code needed for the implementation, and also regarding to the execution time. Figure 1 shows the amount of code needed in R, Java and MATLAB for the same job, and Figure 2 demonstrates the execution times in each of them. Despite of the performance issues, it is obvious that R is a better choice for data-analysis than a general purpose high-level language Java, requiring a lot of programming effort to achieve the same result as with 20 lines of R code. From this perspective, R turns to be also better than MATLAB, as it performs faster.

Consider one more example, shown on Listing 5. This is just an example, illustrating computations performed on data, stored inside a data-frame. It is often convenient to use tabular representation of the data.

One of the most often approaches is to use one of the columns as a measure for grouping, and then perform calculations on the rest of the data.

For example, all the data about the employees in a company can be represented by a data-frame. It may be needed to calculate the average salary or average number of work hours for each position in the company. For these, a grouping based on employee's position should be made first. Then, for each group, the average should be calculated. Of course, it is likely that the algorithms, that need to be applied to grouped data, can be much more complex. Listing 5 illustrates that situation. A data frame with 3 columns and 1000 elements is created. The third column of the data-frame contains values 1 and 2, alternately, so it is a good base for grouping the data into two groups.

The function aggregate can be applied to multiple columns of data-frame, using the list of groups, provided as the second parameter, and applying the function given as the third parameter. This call will apply the anonymous function to the first two columns and group the results, based on the values of the third column. The anonymous function creates a vector of 100 elements for each value, where the elements are the square root of the value.

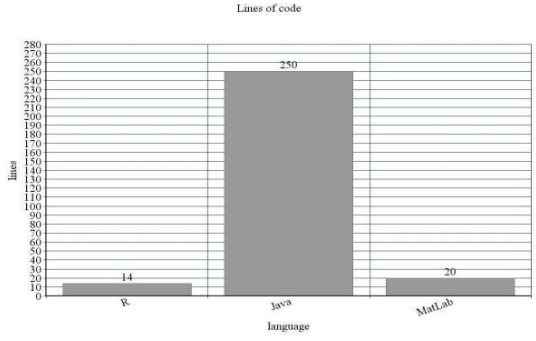

Figure 1. Execution times for time-series mining case- study in R, Java and MATLAB

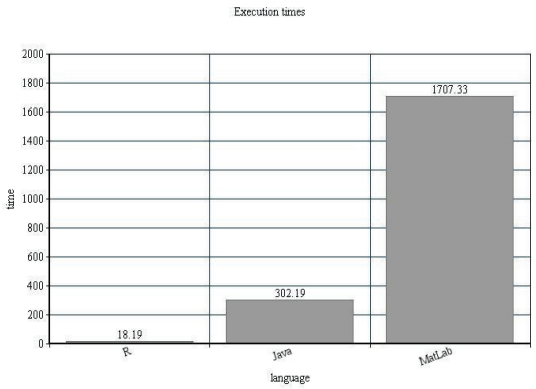

Figure 2. Lines of code needed for time-series mining case study in R, Java and MATLAB

Finally, it calculates the mean value of the numbers from the vector. The final result is a data-frame again with 2 rows and 2 columns.

The example showed the case of applying an operation on a vector for each element of data-frame. This is an implicit use of nested loops. The rules of using vectorized functions were followed, but still the execution time is 49.59 seconds. If the number of elements for vectors is decreased to 50, the timing is 12.28 seconds. So, for doubling the number of elements in vector per data-frame element, the time is 4 times longer.

Based on the examples illustrated, it can be concluded that despite of paying attention to write efficient programs, R expresses serious problems when it comes to the execution time of programs, that rely on working with large data structures, or that require complex calculations in terms of nested loops. It is clear that these concepts impair performance in other languages too, but usually not as much as here.

```
start<-proc.time()
prop1<-1:1000
prop2<-seq(2,2000,2)
prop3<-rep(c(1,2),500)
myData<-data.frame(prop1,prop2,prop3)
res<-aggregate(myData[,1:2],
list(myData$prop3),
               function(x){
                 sapply(rep(sqrt(x),100), mean)
               })
end<-proc.time()
total<-end-start
print(total)
```

Listing 5 – Case study, data-frame data analysis.

Since these cases are very common in practice, the conclusion is that the performance of R is not at an appropriate level, and that there exists a need to find an alternative solution for higher level of efficiency.

V. THE POSSIBILITIES FOR IMPROVEMENT

The need for higher performance R is evident. However, the improvement needs to retain the basic form of the language. Introducing changes to the syntax and semantics or including additional elements in the source code could lead to the loss of simplicity and clarity. Therefore, our idea is to construct a new, more efficient implementation of the language, rather than attempting to improve the existing one.

Building a language from the ground is a demanding undertaking that requires knowledge from different fields, in order to create a language specification, a compiler construction, a set of necessary libraries, an integration of optimizations, and many other aspects.

This complexity can be overcome, by means of modern approaches to build embedded languages within a host language. To achieve this, it is very important to choose a host language that widely supports the implementation of other languages inside it. Domain specific languages (DSLs) are those that are embedded inside other languages. R could be considered as a DSL as well, since it is mainly oriented to statistical calculations, although it is officially not treated so. An appropriate host language should provide a convinient way to introduce the DSLs abstractions, based on the constructions of the host language. Basic features of the host language, as its syntax clarity, extremely influence the process of embedding a new language.

Another important aspect, when building a DSL, is to pay attention to the possibilities for optimization and potentially parallelization on lower levels, without disturbing the form of the DSL. It is also very important to find a right way to overcome the high level of abstraction, as one of the main features of DSLs. Although abstraction simplifies the use of the language, it is in contradiction with the need for efficiency. There are two standard ways to overcome this barrier. The first one is the elimination of abstractions before compiling, so the compiler is not even aware of them. This can be achieved using staging. Staging or multi-stage programming performs a separation of the process of compiling into series of successive phases, which enables code generation. The second approach is to provide domain-specific knowledge to the compiler, by adding new phases of compilation and creating intermediate representations. In recent period, there appeared some approaches that can combine these two mechanisms, through the creation of intermediate representations using staging. Thus, the principles of construction of embedded DSLs, with the inclusion of optimizations, relying on generative programming through staging will be our basic idea while attempting to build a new R implementation.

VI. CONCLUSION

According to described properties of the R language, it turns out that its popularity is justified due to its simplicity and wide range of areas of use. However, when it comes to dealing with problems related to large data sets, efficiency problems arise and disable its use in such applications. We showed some of the aspects, influencing deficiency of R through examples and comparison with other languages.

The objective for further work will be to build a more efficient implementation of the language, to make a

comparative analysis of performance and evaluate the program execution. The described concepts of building DSLs will be the leading ideas. Primarily, it is necessary to identify an appropriate host language with appropriate tools that supports optimizations or generative programming. Such an approach could provide a completely new implementation of the subset of R, with significantly better performance, that could enter into wider use.

ACKNOWLEDGMENT

Results presented in this paper are the result of collaboration between Faculty of Science in Novi Sad and Ecole Polytechnique Federale de Lausanne. The work is partially supported by Ministry of Education, Science and Technological Development of the Republic of Serbia, through project no. ON174023: Intelligent techniques and their integration into wide-spectrum decision support.

REFERENCES

[1] F. Morandat, B. Hill, L. Osvald and J. Vitek, "Evaluating the Design of the R Language", in Proceedings of European conference on Object-Oriented Programming (ECOOP), 2012.

[2] T. Kalibera, P. Maj, F. Morandat and J. Vitek, "A Fast Abstract Syntax Tree Interpreter for R", in Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '14, New York, USA, pp.89-102, ACM, 2014.

[3] D. Eddelbuettel and C. Sanderson, "RcppArmadillo: Accelerating R with High-Performance C++ Linear Algebra". Computational Statistics and Data Analysis, 71, 2014.

[4] H. Wang, P. Wu and D. Padua, "Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-level Specialization", in Proc. CGO, pp.295-295, 2014.

[5] J. Li, X. Ma, S. Yoginath, G. Kora and N. F. Samatova, "Transparent Runtime Parallelization of the R scripting language". Journal of Parallel and Distributed Computing, 71(2), 157-168, 2011.

[6] L. Jiang, P. B. Patel, G. Ostrouchov and F. Jamitzky, "OpenMP-style parallelism in data-centered multi-core computing with R", in Proc. PPOPP, 2012, 335-336

[7] R. Ihaka and R. Gentleman. "R: A Language for Data Analysis and Graphics". Journal of Computational and Graphical Statistics, 5 (3):299-314, 1996.

[8] T. Wurtinger, C. Wimmer, A. Woss, L. Stadler, G. Duboscq, C. Hummer, G. Richard, D. Simon and M. Wolczko, "One VM to Rule Them All", in Proceedings of Onward!, the ACM Symposium on New Ideas in Programming and Reflections on Software, 2013.

[9] R Development Core Team, "R: A language and Environment for Statistical Computing", R Foundation for Statistical Computing, 2011.

[10] R Development Core Team, "The R Language Definition". R Foundation for Statistical Computing, http://cran.r-project.org/doc/manuals/R-lang.html

[11] J. M. Chambers, "Software for data analysis: Programming with R", Springer, 2008

[12] R. Gentleman, et. al., eds. "Bioinformatics and computational biology solutions using R and Bioconductor", Statics for Biology and Health, Springer, 2005.

[13] R. Gentleman and R. Ihaka. "Lexical Scope and Statical Computing". Journal of Computational and Graphical Statistics, 9: 491-508, 2000.

[14] D. Smith. "The R ecosystem". In The UseR Conference 20011, August 2011.

[15] J. Talbot, Z. DeVito, and P. Hanrahan. "Riposite: a trace-driven compiler and parallel VM for vector code in R". In proceedings of Parallel Architectures and Compilation Techniques (PACT), 2012.

[16] L. Tierney. "A byte code compiler for R". University of Iowa, 2015. http://homepage.stat.uiowa.edu/~luke/R/compiler/compiler.pdf

[17] R project. CRAN: The comprehensive R archive network, 2015. http://cran.r-project.org

[18] Bioconductor: Open source software for Bioinformatics, 2015. http://www.bioconductor.org