Statistically Significant Comparative Performance Testing of Julia and Fortran Languages in Case of Runge–Kutta Methods

Migran N. Gevorkyan[1], Anna V. Korolkova[1], Dmitry S. Kulyabov[1,2(✉)], and Konstantin P. Lovetskiy[1]

[1] Department of Applied Probability and Informatics, Peoples' Friendship University of Russia (RUDN University), 6 Miklukho-Maklaya St, Moscow 117198, Russian Federation
{gevorkyan_mn,korolkova_av,kulyabov_ds,lovetskiy_kp}@rudn.university
[2] Laboratory of Information Technologies, Joint Institute for Nuclear Research, 6 Joliot-Curie St., Dubna, Moscow region 141980, Russian Federation

Abstract. In this paper we compare the performance of classical Runge–Kutta methods implemented in Fortran and Julia languages. We use the technique described in technical report by Tomas Kalibera and Richard E. Jones from University of Kent. This technique allows to solve the following problems. 1. The determination of the number of runs required by the program to pass the warm-up stage (e.g. JIT-compilation, memory buffers filling). 2. The determination of the optimal number of levels of the experiment and the number of repetitions at each level for robust testing. 3. The construction of the confidence interval for the resulting average run time. For the numerical experiment we implement 6-th order classical Runge–Kutta methods in both languages in the most similar way. We also study unvectorized versions of our functions. For Julia we tested not only built-in vectorization capabilities, but also external library. For processing the results of measurements Python 3 with Matplotlib, NumPy and SciPy (stats module) were used. We carried out experiments for variety of ODE dimensions (from 2 to 64) and different types of processors. Our work may be interesting not only for the results of comparison of the new Julia language with Fortran, but also for the robust testing method demonstration.

Keywords: Runge–Kutta scheme · Julia language
Fortran language · Performance

The publication has been prepared with the support of the "RUDN University Program 5-100" and funded by Russian Foundation for Basic Research (RFBR) according to the research project No 16-07-00556.

G. Nikolov et al. (Eds.): NMA 2018, LNCS 11189, pp. 400–407, 2019.
https://doi.org/10.1007/978-3-030-10692-8_45

1 Introduction

In this paper, we compare the possibilities of vectorization of operations with arrays in Fortran and Julia [6,7] languages and the application of vectorization for implementation of classical Runge–Kutta schemes [9,11,12]. Average time measurements are made by the method described in papers [1,15]. This technique allows to obtain the statistically significant estimator of average time and determine the optimal number of measurements. In [1,15] the authors described in detail all the steps of their methodology for obtaining statistically significant estimators of the average time of programs execution. They also provided some numerical examples. In the first part of our paper, we will describe how all necessary statistical computations may be implemented by using Python [5,16] with NumPy [4], SciPy [14] and Matplotlib [3]. Our source code is open and available at https://bitbucket.org/mngev/fortran-vs-julia. The second part of the article describes the possibilities of arrays vectorization in Fortran and Julia languages. The programs that we used to compare these two languages are presented. We calculate statistically significant estimators of programs average time execution and the confidence interval for each measurement.

In the third part of the article, we compare the performance of two implementation of classical Runge-Kutta method — in Fortran and Julia languages. SIMD Vectorization is the most natural way to improve the performance of such methods, so the language (strictly speaking the compiler) that implements the best vectorization of actions with small arrays (vectors) will show better performance.

2 The Method of Program Execution Time Measuring

When measuring the execution time of the program the repetition of trials is generally accepted, as well as skipping a certain number of runs for so-called "warm-up". However, the number of trials and the number of runs for warm-up is often determined heuristically and the measurement results may be statistically insignificant.

In our work, we use the [1,15] methodology, which we briefly will describe in this part of paper, focusing on the implementation with Python and stack of scientific libraries.

2.1 The Number of the Experiment's Levels

In our case, the experiment will be understood as execution time measurement for a function, subprogram, program or software complex. Before starting an experiment, one should determine the maximum number of *experiment levels*.

For example, if we want to estimate the execution time of some function in a compiled programming language, we can distinguish three levels of the experiment. The first level is r_1 function calls inside the program, the second

level is r_2 executions of program and the third is r_3 completions of the program source file.

In general, we assume that there is $n+1$ level of repetition of the experiment. At each level i, r_i tests are performed, where $i = 1, \ldots, n+1$. Number of tests at the highest level is r_{n+1}. The described technique is divided into three stages.

2.2 The Results of the Time Measurements

At the first stage, a preliminary experiment with a heuristic choice of the number of levels $n+1$ and the number of tests r_i at each i-th level is carried out. Each test gives a value of the measurement time $X_{j_{n+1}j_n \ldots j_2 j_1}$, where $j_i = 1, \ldots, r_i$. For example, if we measure the time spent on the third call of a function when the program is run for the second time using the fourth compiled executable file, it will be X_{423}. To store the results of measurements we used a multidimensional NumPy array. Every element of this array may be considered independent random number. At each level, a one-dimensional sample of the obtained time measurements is considered.

2.3 First Step: Visual Analysis

After the measurements are completed, one should proceed to a visual evaluation of the data, which will give an opportunity to estimate the number of pre-runs required for the system warm-up. We denote the number of pre-runs for each level by $c_1, c_2, \ldots, c_{r_i}$. For each level i one should draw three plots: run-sequence plot, lag-plot and auto-correlation function plot (ACF-plot).

To plot the autocorrelation graph, the use of the `acorr` function will give an incorrect result, since it uses a different algorithm for calculating autocorrelation in signal processing. So the calculation of autocorrelation must be implemented independently. The result of its work can be displayed on the chart using the `vlines` function.

2.4 Second Step: Biased and Unbiased Estimators

S_i^2 is biased estimator for variance at level i. Despite the cumbersome mathematical formulas, the computation of S_i^2 in terms of NumPy arrays is a trivial task. For example, let us measure time with three levels of experiment. As a result of measurements we obtain a three-dimensional array `X`. To calculate S_i^2, use `mean` function for 2 and 3 dimensions, and then `var` function to find the unbiased sample variance.

3 Vectorization of Action with Arrays in Fortran and Julia

3.1 SIMD Instructions

Most modern processors support SIMD (single instruction stream/multiple data stream) instructions. This means that a single processor core can apply the same

operation to multiple numbers at the same time. To do this, the data should be written to special *vector* registers. For x86 architecture the most widespread technologies are: MMX (MultiMedia eXtension), SEE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) of different revisions. For the measurements we used processors with AVX support, which gives 16 registers with a total volume of 256 bits.

3.2 Support for SIMD Instructions in Fortran and Julia

To measure the efficiency of SIMD capabilities utilization, let's consider several ways to add one array to another in Fortran and Julia languages. Let's begin from Fortran and set three dynamic arrays and then fill them with random numbers. You may also use the `concurrent` statement added to the Fortran 2008 [8,10,13] standard to explicitly point out the loop independence.

There are three options for similar actions in Julia. Two of them are comparable with Fortran. The third option allows using `@simd` and `@inbounds` macros to explicitly specify the usage of SIMD instructions.

3.3 Benchmarking

For Fortran program tests we used GNU Fortran [2] compiler version 7.3.0. To get a file with a report about loops vectorization attempts, it is necessary to add the option `-fopt-info-vec-all=file.log` to compiler. We have implemented pre-mentioned examples as separate pure functions. The Fortran source code is located in the `fortran` directory in the `src` folder. Each function takes numeric arrays as arguments, performs their addition and returns the result as an array. To measure the time required to call functions, we used the intrinsic subroutine `cpu_time`.

Summation was performed over arrays with floating point numbers of single precision (32 bits). The length of the array was calculated from the total volume of vector AVX registers (256 bits). The division by 32 bits gives us exact 8. The time required for 10^4 function calls was measured. Each measurement was repeated 100 times, the program was executed 20 times and compilation was performed also 20 times. In general, it turned out $100 \cdot 20 \cdot 20$ measurements.

Note also that optimization flags `-O1`, `-O2` and `-O3` were used during compilation.

Table 1. Results for Fortran time measurements. Confident interval was calculated for $\alpha = 0.01$

Overall mean				Conf. interval sizes			Optimum		
Function	O1	O2	O3	O1	O2	O3	O1	O2	O3
Iter.	0.000436	0.000405	0.000400	0.000009	0.000010	0.000008	34	41	33
Vect.	0.000246	0.000374	0.000379	0.000011	0.000009	0.000009	27	29	36

The obtained results are summarized in the Table 1. In table one can find the overall mean time of execution for each of two functions, the confidence interval and the optimal number of repetitions for all stages, calculated by the method described above. For clarity, the overall mean time is shown as bar charts in the Fig. 1.

The bar chart clearly shows that when adding arrays without loops and without optimization (flag `-O1`), the compiler automatically applied vectorization. At higher optimization levels, the compiler apply vectorization also to loops. It is noteworthy that the performance of the function with non-index addition excels slightly from more aggressive optimization (`-O2` and `-O3` flags).

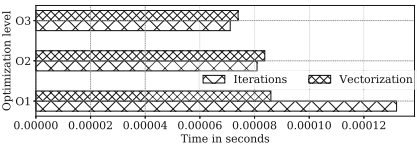

Fig. 1. Overall mean from Table 1

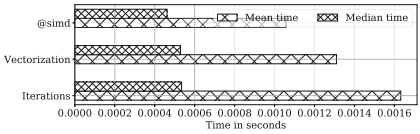

Fig. 2. Overall mean from Table 2

Similar measurements were carried out for programs in Julia. Due to the use of JIT compilation, the experiment levels were reduced from three to two: 20 program launches with 100 measurements at each launch. Also, there are no levels of optimization.

The results of measurements are summarized in the Table 2, and the overall mean time is clearly shown in the form of bar chart 2.

Table 2. Results for Julia time measurements. Confident interval was calculated for $\alpha = 0.01$

	Iterations	Vectorization	`@simd`
Overall mean	0.00163201	0.00131059	0.00105772
Conf. interval sizes	0.0011087924109	0.00087504387	0.00071035994
Optimal	11	11	11

The bar chart shows that the average time of Julia program is almost two orders of magnitude lower than Fortran. However, the median time is less than the average Fortran program execution time by an order of magnitude. If we consider the run-sequence plot, it becomes clear that a some values of measurements differ greatly from the average time. It is these anomalous measurements that lead to an increase in the average time.

As mentioned in the official documentation of the Julia language, the use of non-index operations in the Julia language does not provide any significant increase in productivity (see median time), as at the moment is only a syntactic

sugar. However, the use of the `@simd` macro allows to get slightly better loop performance.

4 Performance Measurements for Runge-Kutta Schemes

Let us first introduce the numerical scheme of the explicit Runge-Kutta method for Cauchy problem [9,12]. Let's consider two smooth functions: $\mathbf{x}(t)\colon [t_0, T] \to \mathbb{R}^N$ and $\mathbf{f}(t, \mathbf{x}(t))\colon \mathbb{R} \times \mathbb{R}^N \to \mathbb{R}^N$, where $[t_0, T] \in \mathbb{R}$. The initial value is $\mathbf{x}_0 = \mathbf{x}(t_0)$. Then the Cauchy problem for a system of N ordinary differential equations is formulated as follows:

$$\begin{cases} \dfrac{\mathrm{d}\mathbf{x}(t)}{\mathrm{d}t} = \mathbf{f}(t, \mathbf{x}), \\ \mathbf{x}(t_0) = \mathbf{x}_0. \end{cases} \tag{1}$$

The explicit s-stage Runge–Kutta method for the Cauchy problem (1) with constant step is given by the following formulas:

$$\begin{aligned} &\mathbf{k}^1 = \mathbf{f}(t_m, \mathbf{x}_m), \\ &\mathbf{k}^2 = \mathbf{f}(t_m + c^2 h, \mathbf{x}_m + h a_1^2 \mathbf{k}^1), \\ &\dots \\ &\mathbf{k}^s = \mathbf{f}\big(t_m + c^s h, \mathbf{x}_m + h(a_1^s \mathbf{k}^1 + a_2^s \mathbf{k}^2 + \dots + a_{s-1}^s \mathbf{k}^{s-1})\big), \\ &\mathbf{x}_{m+1} = \mathbf{x}_m + h(b_1 \mathbf{k}^1 + b_2 \mathbf{k}^2 + \dots + b_{s-1} \mathbf{k}^{s-1} + b_s \mathbf{k}^s) \end{aligned}$$

For this scheme it is difficult to build a parallel algorithm based on processes, since each $\mathbf{k}^i, i = 1, \dots, s$ depends explicitly on all previous ones and cannot be calculated independently. However, vectorization with SIMD instructions can give a performance benefit, since expression $a_1^i \mathbf{k}^1 + a_2^i \mathbf{k}^2 + \dots + a_{s-1}^i \mathbf{k}^{i-1}, i = 1, \dots, s$ can be efficiently vectorized at each stage.

We implemented an explicit numerical scheme with order $p = 6$ and stage $s = 7$ in Fortran and Julia languages in the most similar way (the function is called `RKp6`). In Fortran, we set the coefficients of the method as separate variables with the `parameter` attribute, and in Julia we used the `const` statement. This will allow Fortran and Julia compiles to optimize the program, as the values of the coefficients will be known at the compilation stage and can not be changed during the program run time.

For testing we used simple linear oscillator equation:

$$\frac{\mathrm{d}x_1(t)}{\mathrm{d}t} = -x_2, \ \frac{\mathrm{d}x_2(t)}{\mathrm{d}t} = x_1,$$

on the time interval $[0, 10]$ with initial value $\mathbf{x} = (1, 1/2)^T$ and with the method step $h = 10^{-4}$. Was performed 1000 calls to the function `RKp6`, with 10 runs of programs (for Julia and for Fortran). We didn't repeat the compilations for Fortran, because the estimator S_3^2 showed that this level of experiment does not have a statistically significant effect on the result.

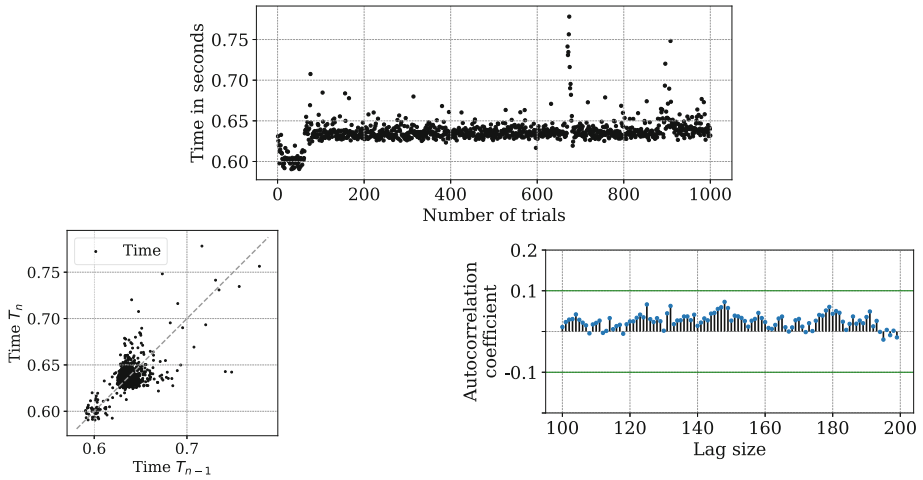

Fig. 3. The run-sequence plot, lag-plot and ACF-plot for Julia Runge-Kutta program measurements.

The Fig. 3 shows three plots required for visual estimation of measurements. The first run-sequence plot shows that approximately the first 100 measurements yielded results that were significantly different from the next ones, so we had to discard them. Moreover, the autocorrelation coefficient for the first 100 measurements was also much higher than the allowed values $[-0.1, 0.1]$. According to lag-plot, the measured values are distributed randomly and do not form any regular structures.

Fortran program was compiled with different optimization flags. The vectorization report showed that the compiler found beneficial to vectorized the addition of $\mathbf{k}^i$ inside the functions. As in our previous measurements, the Julia program was much slower (Fig. 4).

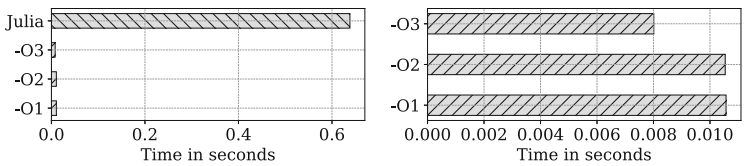

Fig. 4. Fortran and Julia Runge–Kutta performance comparison

5 Conclusion

We have measured the performance of Fortran and Julia languages in tasks that benefit from the use of vector instructions of modern processors. In addition, we briefly described the methodology [1,15], which we used during for our measurements.

Despite the fact that Julia showed worse results than Fortran program, it is necessary to make a number of notes in favor of Julia.

- Julia Language is still under heavy development and major changes may be expected in the syntax and the individual aspects of performance.
- We have performed measurements only for a specific array vectorization task, so for other tasks the performance difference may not be so significant.
- We have not considered the external library for static arrays for Julia, which could potentially bring performance boost.
- Julia is a dynamic language, so in most cases it will win against Fortran in development speed.

The source code of the examples and additional performance measurements can be found by link https://bitbucket.org/mngev/fortran-vs-julia.

References

1. Rigorous Benchmarking in Reasonable Time. ACM, New York, June 2013
2. Gnu fortran (2018). https://gcc.gnu.org/fortran/
3. Matplotlib home site (2018). https://matplotlib.org/
4. Numpy home site (2018). http://www.numpy.org//
5. Python home site (2018). https://www.python.org/
6. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Rev. **59**(1), 65–98 (2017). https://doi.org/10.1137/141000671
7. Bezanson, J., Karpinski, S., Shah, V.B., Edelman, A.: Julia: A Fast Dynamic Language for Technical Computing, September 2012
8. Brainerd, W.S.: Guide to Fortran 2008 Programming. Springer, London (2015). https://doi.org/10.1007/978-1-4471-6759-4
9. Butcher, J.: Numerical Methods for Ordinary Differential Equations, 2nd edn. Wiley, New Zealand (2003)
10. Chapman, S.J.: Fortran for Scientists and Engineers. McGraw-Hill Education, New York (2018)
11. Gevorkyan, M.N., Velieva, T.R., Korolkova, A.V., Kulyabov, D.S., Sevastyanov, L.A.: Stochastic Runge–Kutta software package for stochastic differential equations. In: Zamojski, W., Mazurkiewicz, J., Sugier, J., Walkowiak, T., Kacprzyk, J. (eds.) Dependability Engineering and Complex Systems. AISC, vol. 470, pp. 169–179. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39639-2_15
12. Hairer, E., Nørsett, S.P., Wanner, G.: Solving Ordinary Differential Equations I, 2nd edn. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78862-1
13. Hanson, R.J., Hopkins, T.: Numerical Computing With Modern Fortran. SIAM, Philadelphia (2013)
14. Jones, E., Oliphant, T., Peterson, P., et al.: SciPy: Open source scientific tools for Python (2001). http://www.scipy.org/
15. Kalibera, T., Jones, R.E.: Quantifying Performance Changes with Effect Size Confidence Intervals. Technical report 4–12, University of Kent, June 2012
16. Rossum, G.: Python reference manual. Technical report, Amsterdam, The Netherlands (1995)