

Parallelizing the code of the Fokker-Planck equation solution by stochastic approach in Julia programming language

Anna Wawrzynczak*,†

*Institute of Computer Sciences, Siedce Univeristy
ul.3 Maja 54, Siedlce, Poland
e-mail: awawrzynczak@uph.edu.pl

†National Centre for Nuclear Research
ul A.Soltana 7, Świerk-Otwock, Poland

Abstract—Presenting a reliable physical simulation requires very often use of the supercomputers and models run for many days or weeks. The numerical computing is divided into two groups. One uses highly efficient low-level languages like Fortran, C, and C++. The second applies high-level languages like Python or Matlab, being usually quite slow in HPC applications. This paper presents the application of the relatively new programming language Julia, advertised as the as "a high-level, high-performance dynamic programming language for numerical computing". We employ Julia is to solve the Fokker-Planck equation by the stochastic approach with the use of the corresponding set of ordinary differential equations. We propose the method of parallelizing the algorithm with use of the distributed arrays. We test the speedup and efficiency of the given code with use of the cluster set at the Świerk Computing Centre and show that Julia is capable of achieving a good performance.

I. INTRODUCTION

CHOICE of the programming language for implementation of the mathematical model is quite a key factor for its future performance. Scientists routinely run simulations on millions of cores in distributed environments. Moreover, this choice is often influenced by the difficulty level of the language and time that has to spend on code production and its parallelization.

The main high-performance computing programming languages are statically compiled language such as Fortran, C, and C++, in conjunction with OpenMP/MPI. The reasons are their interoperability and efficiency regarding the ability to use all available compute resources while limiting memory usage. These languages are compiled off-line and have strict variable typing, allowing advanced optimizations of the code to be made by the compiler. Taking above into account one can think that the choice of a programming language is natural. However, it seems that writing HPC code is getting more complicated because today's projects often require a combination of messaging (MPI), threads (OpenMP), and accelerators (Cuda, OpenCL, OpenACC). This causes that creating an HPC code seems to be getting more difficult

This work is supported by The Polish National Science Centre grant awarded by decision number DEC-2012/07/D/ST6/02488

instead of more straightforward. Of course, it is acceptable from the point of view of computer scientists or developers, but for the scientists from other fields more imperative is to get a relatively quick result to confirm/reject their theories or models.

The answer to that problems are the modern interpreted languages. In these languages, programs may be executed from source code form, by an interpreter. The advantages of this class are that they are often easier to implement in interpreters than in compilers, include platform independence, dynamic typing and dynamic scoping. The disadvantage is that they are usually slower than the compiled one. Examples of languages of this type are, e.g., Octave, Scilab, R, Mathematica, and Matlab. This category of languages is also known as dynamic languages or dynamically typed languages. In these programming languages, programmers write simple, high-level code without any mention of types like `int`, `float` or `double` that pervade statically typed languages such as C and Fortran. The overview of dynamical languages in comparison with the more traditional languages is presented in [1].

II. JULIA PROGRAMMING LANGUAGE

In this section we would like to introduce a Julia language, in which was implemented the algorithm presented in this paper. This will not be a tutorial, only some functions and issues useful in the parallel implementation of the proposed algorithm will be presented. For a full information we refer to [2].

Julia is a new programming language that is designed to address the problem of the low performance of dynamic languages [3]. In benchmarks reported by its authors, Julia performed within a factor of two of C on a set of standard basic tasks. Julia is advertised as a high-level, high-performance dynamic programming language for numerical computing. It provides a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library. Julia's Base library, written mainly in Julia itself, also integrates mature, best-of-breed open source C and Fortran libraries for linear algebra, random

number generation, signal processing, and string processing.

Julia is a free open source language that can be downloaded from the website [2]. The core of the Julia implementation is licensed under the MIT license. The language can be built as a shared library so that users can combine Julia with their own C/Fortran code or proprietary third-party libraries. The GitHub repository of Julia source code, packages, and documentation can be downloaded from website [4].

Users interact with Julia through a standard REPL (real-eval-print loop environment) such as Python, R, or MATLAB), by collecting commands in a .jl file, or by typing directly in a Jupyter (Julia, PYThon, R) notebook [5]. Julia syntax is "Matlab like" and allows uncomplicated expression of mathematical formulas. Julia uses just-in-time (JIT) compilation [6], [7] using the Low-Level-Virtual-Machine (LLVM) compiler framework [8]. JIT compilers attempt to compile at run-time by inferring information not explicitly stated by the programmer and use these inferences to optimize the machine code that is produced.

A. Parallel Computing in Julia

Julia has some built-in primitives for parallel computing at every level: vectorization (SIMD), multithreading, and distributed computing. At the lower level, Julia's parallel processing is based on the idea of remote references and remote calls. Remote call send a request to run a function on another processor, while remote reference create an object used to refer to objects stored on a particular processor. The parallel programming in Julia is quite intuitive. Starting the Julia in command line with `julia -p n` provides `n` worker processes on the local machine. It makes sense for `n` to be equal the number of CPU cores on the machine. In the script the additional `n` processes can be added by function `addprocs(n)` and removed by `rmprocs()`. Number of active processes can be listed by `workers()`.

Consider a parallel computation of the pi value by the Monte Carlo method. This calculation contains generating random numbers between 0 and 1 and ultimately calculating the ratio of the ones lying in inside the unit circle to those that don't.

```
addprocs(2) #add 2 Julia worker processes

function parallel_PI(n)
    in_circle = @parallel (+) for i in 1:n
        # work parallelizing
        x = rand()
        y = rand()
        Int((x^2 + y^2) < 1.0)
    end
    return (in_circle/n) * 4.0
end
```

The above function execution is:

```
parallel_PI(10000)
```

`@parallel` is for parallelizing loops. Julia offloads task to its worker processes that compute the desired output and send them back to the Julia master, where the reduction is performed. Arbitrary pieces of computation can be assigned to different worker processes through this one-sided communication model.

The algorithm presented in section III requires using large arrays for storing the pseudoparticles position in time of simulation. In such cases the distribution of the array between the workers seems to be a good solution. A distributed array is logically a single array, but its fragments are stored on several processors. Such approach allows making a matrix operation the same like with local arrays, making the parallelism almost invisible for the user. In some cases, it is possible to obtain useful parallelism just by changing a local array to a distributed array. Moreover, it makes possible to use an array of a size that wouldn't be possible to create in memory of one master process.

In Julia distributed arrays are implemented by the `DArray` type, which from version 0.4 has to be imported as the `DistributedArrays.jl` package from [9]. A `DArray` has an element type and dimensions just like a Julia array, but it also needs an additional property: the dimension along which data are distributed. Distributed array can be created in a following way:

```
julia>addprocs(4)
julia>@everywhere using DistributedArrays
julia>Tab1=drandn(8,8,4)
julia>Tab2=dfill(-1,8,8,4)
julia>Tab3=dzeros(8,8,4)
julia>Tab4=dzeros((8,8),workers()[1:4],[1,4])
```

In the above code, the four workers are started. Macro `@everywhere` allow precompiling the `DistributedArrays` on all processors. In the declaration of `Tab1`, `Tab2`, `Tab3` the distribution of this 8×8 arrays between four processors will be automatically picked. Random numbers will fill `Tab1`, `Tab2` will be filled with number `-1`, and `Tab3` with zeros. In the definition of the `Tab4` user can specify which processes to use, and how the data should be distributed. The second argument specifies that the array should be created on the first four workers. The third argument specifies on how many pieces chosen dimension should be divided into. In this example, the first dimension will not be divided, and the second dimension will be divided into 4 pieces. Therefore each local chunk will be of size $(8,2)$. The product of the distribution array must equal the number of workers.

This way of parallelization is quite convenient because when dividing data among a large number of processes, one often sees diminishing gains in performance. Placing `DArray` on a subset of processes allows numerous `DArray` computations to happen at once, with a higher ratio of work to communication on each process. Method `indexes` allow checking how the array is distributed. For example output of instruction

```
julia>Tab1.indexes
```

can be following

```
2x2x1 2x2x1 Array{Tuple{UnitRange{Int64},
  UnitRange{Int64},UnitRange{Int64}},3}:
[:, :, 1] =
(1:4,1:4,1:4) (1:4,5:8,1:4)
(5:8,1:4,1:4) (5:8,5:8,1:4)
```

We see that array is divided into four parts versus a number of rows and columns.

Other useful operations on distributed arrays are:

- `distribute(a::Array)` converts a local array to a distributed array,
- `localpart(a::DArray)` obtains the locally-stored portion of a DArray,
- `myindexes(a::DArray)` gives a tuple of the index ranges owned by the local process,
- `convert(Array, a::DArray)` brings all the data to the local processor.

When a DArray is created (usually on the master process), the returned DArray object stores information on how the array is distributed. When the DArray object on the master process is garbage collected, all participating workers are notified and `localparts` of the DArray freed on each worker. Since the size of the DArray object itself is small, a problem arises as `gc` on the master faces no memory pressure to collect the DArray immediately. This results in a delay of the memory being released on the participating workers. Therefore it is required to explicitly call `close(d::DArray)` after user code has finished working with the distributed array [9].

III. THE ALGORITHM FOR NUMERICAL SOLUTION OF THE FOKKER-PLANCK EQUATION

The Fokker-Planck equation (FPE) arises in a wide variety of natural science, including solid-state physics, quantum optics, chemical physics, theoretical biology and astrophysics. The FPE was first utilized by Fokker and Planck to describe the Brownian motion of particles e.g. [10]. In this paper we will use this equation to describe the transport of cosmic rays (CR) throughout the heliosphere, originating from outer space and reaching the Earth (e.g., [11]). The transport of CR particles is usually described by the Parker transport equation (PTE) [12]. The difficulty of the numerical solution of this type equations increases with the problem dimension. Reason is the instability of the numerical schemes like finite-differences (e.g. [13]) and finite-volume in the higher dimensions. In consequence, to ensure the scheme stability and convergence the density of numerical grid must be improved, increasing the computational complexity. To overcome this problem the stochastic methods can be applied. In that case the PTE should be rewritten in the form of the FPE (for details see, e.g. [14], [15]), as:

$$\frac{\partial \hat{f}}{\partial t} = \vec{\nabla} \cdot [\vec{\nabla} \cdot (K^T \hat{f})] - \vec{\nabla} \cdot [(\vec{\nabla} K^T + U) \cdot \hat{f}] + \frac{1}{3} \frac{\partial}{\partial R} [(\hat{f} R (\vec{\nabla} \cdot \vec{U})) - L \cdot \hat{f}]. \quad (1)$$

Where $\hat{f} = \hat{f}(\vec{r}, R, t)$ is an omnidirectional distribution function depending on spherical coordinates $\vec{r} = (r, \theta, \varphi)$, r - radial distance, θ - heliolatitudes, φ - heliolongitudes;

magnetic rigidity R and time t . $R = \frac{Pc}{q}$, where P is momentum, c speed of light, $q = Ze$, Z charge number of nucleus and e unit charge; \vec{U} is the solar wind velocity, K is the anisotropic diffusion tensor, K^T its transpose; L is the linear factor.

Applying the Ito stochastic integral we can bring the solution of the Eq. (1) to the solution of the set of stochastic ordinary differential equations (SDEs) being the exact equivalence of the FPE (e.g., [16]). Details of this procedure are given in ([14], [15] and references therein). Accordingly, the transport of CR in the 2D heliocentric coordinate system, in which $\vec{r} = (r, \theta)$, can be described by the following SDEs:

$$\begin{aligned} dr(t) &= \left(\frac{2}{r} K_{rr}^S + \frac{\partial K_{rr}^S}{\partial r} + \frac{ctg\theta}{r} K_{\theta r}^S + \frac{1}{r} \frac{\partial K_{\theta r}^S}{\partial \theta} + U + v_{d,r} \right) \cdot dt \\ &\quad + [B \cdot dW]_r \\ d\theta(t) &= \left(\frac{K_{r\theta}^S}{r^2} + \frac{1}{r} \frac{\partial K_{r\theta}^S}{\partial r} + \frac{1}{r^2} \frac{\partial K_{\theta\theta}^S}{\partial \theta} + \frac{ctg\theta}{r^2} K_{\theta\theta}^S + \frac{1}{r} v_{d,\theta} \right) \cdot dt \\ &\quad + [B \cdot dW]_\theta \\ dR(t) &= -\frac{R}{3} (\vec{\nabla} \cdot U) \cdot dt. \end{aligned} \quad (2)$$

In set of Eqs. (2) \vec{v}_d the drift velocity calculated as: $v_{d,i} = \frac{\partial K^A}{\partial x_j}$, where K^A is the antisymmetric part of the anisotropic diffusion tensor of the CR particles $K = K^S + K^A$ and $K^T = K^S - K^A$, containing the symmetric K^S and antisymmetric K^A parts given in [18]. The stochastic terms contain an element $d\vec{W}$ which is the increment of Wiener process guiding the stochastic motion of pseudoparticles in given dimension. B_{ij} , ($i, j = r, \theta$) is a matrix given in [15]. We discretize the set of Eqs. (2) with the unconditionally stable Euler-Maruyama [17] scheme. Nevertheless, the Eqs. (2) does not contain the linear factor L . Thus its solution is not synonymous with a solution of Eq. 1. In numerical realization we introduced weight W in which a linear factor L is taken into account according to formula:

$$W = \exp\left(-\int_0^t L(t) dt\right). \quad (3)$$

Consequently, the \hat{f} function value is expressed as a weighted average having the following form:

$$\begin{aligned} \hat{f}(\vec{r}, R) &= \frac{1}{N_f} \sum_{n=1}^{N_f} f_{LIS}(R) \cdot W = \\ &= \frac{1}{N_f} \sum_{n=1}^{N_f} f_{LIS}(R) \cdot \exp\left(-\sum_{m=1}^M L_{m} \cdot \Delta t\right). \end{aligned} \quad (4)$$

$L = -\frac{2}{3} \vec{\nabla} \cdot U$ is the linear factor visible in Eq. 1, N is the total number of simulated pseudoparticles, N_f is the number of pseudoparticles reaching the position \vec{r} and M is the number of time steps. Function $f_{LIS}(R)$ denotes the \hat{f} value at the boundary.

During the simulation pseudoparticles are initialized at the region (heliosphere) boundary with the initial rigidity drawn by the rejection sampling algorithm from $f_{LIS}(R)$ distribution; then their trajectories are traced in conjunction with changes of their rigidity R . The position and rigidity of each pseudoparticle in every time step must be stored to find the value of the distribution function $\hat{f}(\vec{r}, R)$ in each (required) point of the region. The pseudoparticle motion is

terminated when it reaches the inner/outer boundary with respect to the radial distance or when the time for simulation finishes. As far as the probability that statistically enough number of pseudoparticles will reach the single point is near to zero, we use the bins instead of the points. Thus, to find the numerical solution of FPE it is necessary to apply the binning procedure, i.e., discretized the 3D domain over all spatial variables: (r, θ) and R . Then for each binning unit $[r \pm \Delta r] \times [\theta \pm \Delta \theta] \times [R \pm \Delta R]$, we integrate the trajectories of pseudoparticles traveling through considered bin according to Eq. 4. The binning procedure is the most time consuming from the computational point of view.

IV. JULIA PARALLEL CODE FOR THE SOLUTION OF FPE

The code for the numerical solution of the set of Eqs. (2) was realized in Julia v 0.6.2 [2]. The stochastic method solution of FPE is quite easy to parallelize versus the number of simulated pseudoparticles, which can be simulated independently. To do this large arrays are required for storing the pseudoparticles position, rigidity and weight in subsequent time steps. A natural way to obtain parallelism is to distribute arrays between many processors. This approach combines the memory resources of multiple machines, allowing to create and operate on arrays that would be too large to fit on one machine. Each processor operates on its own part of the array, making possible a simple and quick distribution of task among machines. A distributed array is logically a single array, but its fragments are stored on several processors. Such approach allows making a matrix operation the same like with local arrays, making the parallelism almost invisible for the user. This way was written the parallel program solving the Eq. 1 by the method described in detail in Section III.

The proposed construction of Julia code solving FPE by the stochastic approach described in section III is given in Algorithm 1.

Command in first line calls the required number of CPUs. In lines 5-9 the tables storing the pseudoparticles position and rigidity are distributed among the processes via the `DistributedArrays` package. The distribution is done versus the second dimension, i.e. a single processor covers array of size $n \times (m/WN)$. The simulation of pseudoparticles motion is done by the function `SEQ()` given in lines 12-36. The pseudoparticle initial position and rigidity is set in line 16-18; its position is changed accordingly to the equation 2 (lines 15-19) including the Wiener process defined in line 20-21. The initial weight is set in line 19, while its change is set in line 31. The simulation is performed until all pseudoparticles meet the termination conditions (line 33). In lines 40-52 is defined the function `PRL()` which runs the function `SEQ()` in parallel on different workers owned by the distributed arrays. The `@spawnat` macro evaluates the expression in the second argument on the process specified by the first argument. Function `pmap()` transform collection `out` by applying `fetch` to each element using available workers and tasks. The actual launch of function `PRL()` is done in line 55.

Algorithm 1 Draft of Julia parallel code for FPE solution

```

1  addprocs(number_of_processors)
2  n=number_of_time_steps;
3  m=number_of_pseudoparticles;
4  WN=length(workers());
5  @everywhere using DistributedArrays
6  r=dzeros((n,m),workers()[1:WN],[1,WN])
7  T=dzeros((n,m),workers()[1:WN],[1,WN])
8  R=dzeros((n,m),workers()[1:WN],[1,WN])
9  W=dzeros((n,m),workers()[1:WN],[1,WN])
10 #####
11
12 @everywhere function SEQ(n,m,r,T,R,W)
13   for j = 1 : m
14     #defining pseudoparticles
15     #initial characteristics in t=0;
16     r[1,j]=...;
17     T[1,j]=...;
18     R[1,j]=...;
19     W[1,j]=1;
20     dWr=Wiener(n);#generating the Wiener
21     dWt=Wiener(n);#processes
22     for i = 1 : n-1
23       #calculation of the dr, dT,
24       #dR, dW according to Eqs.2
25       ...
26       #calculation of new pseudopart.
27       #characteristics
28       r[i+1,j]=r[i,j]+dr[i,j];
29       T[i+1,j]=T[i,j]+dT[i,j];
30       R[i+1,j]=R[i,j]+dR[i,j];
31       W[i+1,j]=W[i,j]*exp(-Lf*dt);
32       boundary_verification;
33       termination_verification;
34     end
35   end
36 end
37
38 #####
39
40 function PRL(n,m,r,T,R,W)
41   P=length(procs(r))
42   Nlocal=[size((r.indexes)[w][1],1)
43           for w=1:P]
44   Mlocal=[size((r.indexes)[w][2],1)
45           for w=1:P]
46   out=[(@spawnat(procs(r))[w]
47         SEQ(Nlocal[w],Mlocal[w],
48            localpart(r),localpart(T),
49            localpart(R),localpart(W)))
50        for w=1:P]
51   pmap(fetch,out)
52 end
53 #####
54
55 @time PRL(n,m,r,T,R,W) #actual run
56 @time PRLBin(n,m,r,r_bin,T,T_bin,R,R_bin,W)

```

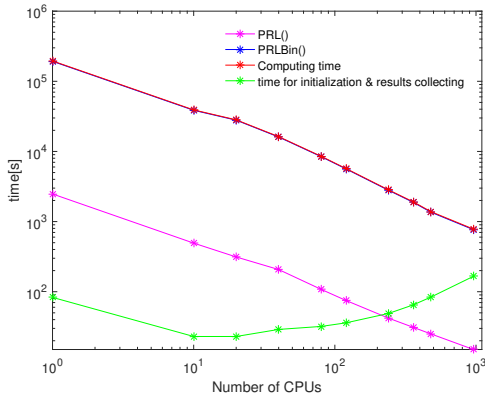


Fig. 1. The time of the FPE solution in Julia ver 0.6.2 on CIŚ cluster. Figure in log-log scale.

To obtain the value of the distribution function we have to run the function `PRLBin()` which search for pseudoparticles falling into the bins $[r \pm \Delta r] \times [\theta \pm \Delta\theta] \times [R \pm \Delta R]$ and apply the Eq. 4 to get the $f(\vec{r}, R)$ value in each bin. The macro `@time` measures the performance of the function run by returning the calculation time and amount of allocated memory.

The launching, management, and networking of Julia processes into a cluster can be done via `ClusterManager.jl` package [19]. It supports different job queue systems commonly used on computer clusters as Slurm, Sun Grid Engine, and PBS. However, this package doesn't support the Torque system installed on CIŚ cluster used to perform simulations presented in this paper. In such case, the distribution can be done directly via the `machinofile`. The sample `.PBS` file that is send to the queue can have a form:

```
#!/bin/bash
#PBS -N task_name
#PBS -l nodes=N :ppn=P
#PBS -q queue_name
cd $PBS_O_WORKDIR
julia --machinofile $PBS_NODEFILE
/mnt/home/user_catalogue/task_code.jl
```

In above-presented script the number of required nodes is equal to N , and number of CPU's per node to P . Thus the $N * P$ workers will be allocated to the job. The names of all nodes the job has allocated, with an entry for every CPU will be saved to the `nodefile`. Thus Julia will read the number of workers from that file, so calling the `adprocs()` in the first line of Algorithm 1 should be omitted.

A. Julia performance

The presented algorithm complexity is $O(n)$ and depends on the number of pseudoparticles and number of time steps. To obtain the reliable results and achieve the satisfactory statistics

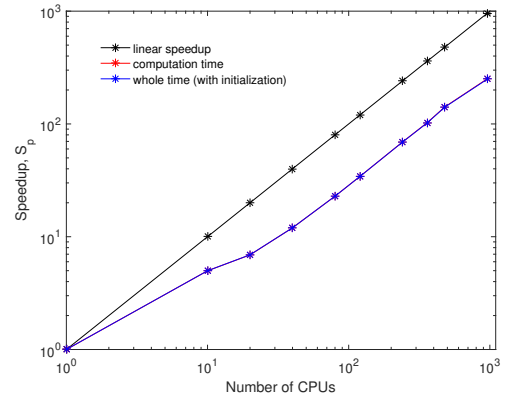


Fig. 2. The calculated speed-up of the FPE solution in ver Julia 0.6.2 on CIŚ cluster. The black line shows the case of ideal speed-up. Figure in log-log scale.

in each 2D heliosphere bin we have to run at least 2 millions of pseudoparticles from the heliosphere boundary. In this paper we do not focus on the results of the physical model, interested reader we refer to the following papers [14], [15]. Here we would like to test the performance of above presented parallel Julia code on the HPC cluster. We have employed the CIŚ machine with characteristic given in Table I.

We have run series of simulations changing the number of CPUs in the range from 1 up to 960. Used model was simplified, as far as it should be run on the one CPU. We assumed the simulation time to be equal to $t = 225$ days with a time step $\Delta t = 1$ hour and a number of simulated pseudoparticles $m = 7200$.

We have analyzed the results accordingly to Amdahl's model [20] that assumes the problem size does not change with the number of CPUs and wants to solve a fixed-size problem as quickly as possible. The model is also known as speedup, which can be defined as the maximum expected improvement to an overall system when only part of the system, is improved. We have used for evaluating the performance of the parallel code the parallel runtime, the speedup S_p calculated as:

$$S_p = \frac{T_s}{T_p}, \quad (5)$$

where T_s is sequential runtime using one CPU, and T_p is runtime using p -number of CPUs. We have also estimated the efficiency E_p as:

$$E_p = \frac{T_s}{p * T_p}. \quad (6)$$

We have run calculations assuming the job distribution over the 1, 10, 20, 40, 80, 120, 240, 360, 480, and 960 CPUs. The number of used CPU has a direct impact on number of pseudoparticles simulated on single CPU, i.e. 7200, 720, 360, 180, 90, 60, 30, 20, 15 and 7, respectively. The results of parallel runtime for the computing the whole simulation, particular functions and the time required for initialization and data collecting present Fig. 1. The computing time includes

TABLE I
THE USED CIŚ MACHINE CHARACTERISTICS.

Feature	Specification
Model	Intel S2600TP
CPU	Intel Xeon(R) CPU E5-2680v2
CPU frequency	2.8Hz, up to 3.6GHz in turbo mode
CPUs per node	40
RAM per node	128 GB, DDR3
Interconnect	1 x Ethernet (1Gbit/sec per port)

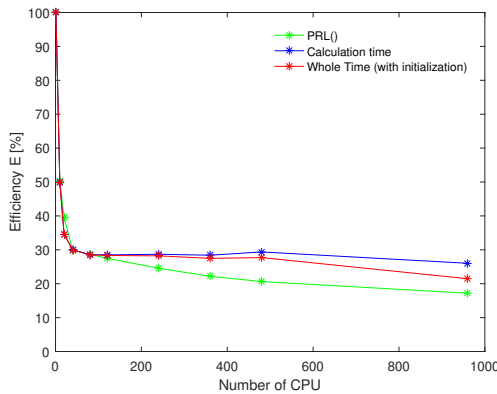


Fig. 3. The efficiency of the FPE solution in ver Julia 0.6.2 on CIŚ cluster.

both the `PRL()` and `PRLBin()` functions parallel execution times. The binning procedure has the most substantial contribution to computing time. We see that the execution time decreases with CPUs number quite steady, simultaneously, the time for initialization and collecting the data increases, because more communication between the nodes is required. The Fig. 2 presents the speedup. We can see that the presented method of parallelization gives a good sublinear speedup. The speedup is behaving very stable and follows a straight line starting from 1 node (40 CPU). Up to one node, the line declines from a straight line. The reason might be that the other tasks overloaded the free CPUs consuming large memory. The corresponding efficiency rate presents the Fig. 3. The efficiency curve shows a negative gradient as the efficiency reduces with an increased number of processors. It stabilizes starting from 80 CPUs up to 480 CPUs at level of 28%, then decreases up to 21%. This metric measures the effectiveness of parallel algorithm concerning the computation time. These results show also that the application of the parallelization based on the distributed arrays is quite efficient in Julia.

V. SUMMARY

We have implemented the proposed method in the new high-level Julia programming language. Parallelization of the code is based on decomposing the problem into a subset of independent tasks versus the number of simulated pseudoparticles. We recognized Julia as a very suitable intuitive

tool for parallel implementation. We have analyzed the HPC performance of Julia as the speedup and efficiency with the use of the CIŚ cluster. We can conclude that the performance of Julia code utilizing the distributed arrays is quite good. Moreover, application of the Julia built-in parallelization methods based on remote references and remote calls does not require from the user much effort or additional work/knowledge on serialization and message passing between workers. These features allow recommending Julia in HPC calculations in the cases when results should be archived relatively quickly and a small amount of time can be taken for the code parallelization.

ACKNOWLEDGMENT

Calculations were performed at the Świerk Computing Centre being a part of the National Centre for Nuclear Research.

REFERENCES

- [1] Rei, L., Carvalho, S., Alves, M., Brito, J., A look at dynamic languages, *Tech. report*, 2007, Faculty of Engineering University of Porto.
- [2] The Julia Language <https://julialang.org/>
- [3] Bezanson J. et. al., Julia: A Fresh Approach to Numerical Computing, *SIAM Review*, vol. 59, 1 (2017) 65-98
- [4] GitHub <https://github.com/JuliaLang/julia>
- [5] The Jupyter Project, <http://jupyter.org/>
- [6] J. Bezanson, Abstraction in Technical Computing, Ph.D. thesis, Massachusetts Institute of Technology, MA, 2015.
- [7] Bezanson J., S. Karpinski, V. B. Shah, and A. Edelman, Julia: A Fast Dynamic Language for Technical Computing, preprint, arXiv:1209.5145 [cs.PL], 2012.
- [8] Lattner C., Adve V., LLVM: A compilation framework for lifelong program analysis & transformation. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, 75
- [9] Distributed Arrays Packadge <https://github.com/JuliaParallel/DistributedArrays.jl>
- [10] Risken, H., *The Fokker-Planck Equation Method of Solution and Applications*, Springer-Verlag, Berlin, Heidelberg, 1989
- [11] Moraal H., Cosmic-Ray Modulation Equations, *Space Science Reviews*, 2013, vol. 176, 299-319 doi:10.1007/s11214-011-9819-3
- [12] Parker E. The passage of energetic charged particles through interplanetary space, *Planetary and Space Science*, 1965, vol. 13, 9-49
- [13] Wawrzynczak A., Alania M.V., Numerical Solution of the Time and Rigidity Dependent Three Dimensional Second Order Partial Differential Equation, *Lecture Notes in Computer Science*, 2010, vol. 6067, pp. 105-114, doi: 10.1007/978-3-642-14390-8_12
- [14] Wawrzynczak A., Modzelewska R and Gil A, Stochastic approach to the numerical solution of the non-stationary Parker's transport equation, *Journal of Physics: Conference Series*, 2015, vol. 574, 012078, doi:10.1088/1742-6596/574/1/012078
- [15] Wawrzynczak A., Modzelewska R and Gil A, The algorithms for forward and backward solution of the Fokker-Planck equation in the heliospheric transport of cosmic rays, *Lecture Notes in Computer Science*, 2018, vol. 10777, 14-23, doi:10.1007/978-3-319-78024-5-2
- [16] Gardiner C.W., *Handbook of stochastic methods. For physics, chemistry and the natural sciences*, Springer Series in Synergetics, 2009
- [17] Kloeden P E, Platen E, Schurz H., *Numerical solution of SDE through computer experiments*, Springer-Verlag Berlin Heidelberg, 1992
- [18] Alania M. V., Stochastic Variations of Galactic Cosmic Rays, *Acta Physica Pol. B*, 2002, vol. 33(4), 1149-1166
- [19] Cluster Manager Packadge <https://github.com/JuliaParallel/ClusterManagers.jl>
- [20] Amdahl, G.M., Validity of the single-processor approach to achieving large scale computing capabilities, in: Proc. Am. Federation of Information Processing Societies Conf., AFIPS Press, (1967) 483-485.