Seminar: Newest Trends in High-Performance Data Analytics

The Julia programming language

and its suitability for HPC

Anna Kahle

September 22, 2022

Supervisor: Marcus Merz

Contents

1 Introduction

In this report I want to discuss whether Julia is suitable to be used for high-performance computing as of now. In the second chapter I will give a short introduction into Julia, its style and performance, before talking about parallel computing in the next chapter. In chapter four I will talk about HPC computing in Julia, focusing on performance and real-world examples. Then I want to conclude by examining the stand of the art.

2 Julia in general

Julia is a rather new programming language. It was developed in 2009 by Dr. Vilah Shah, Prof. Alan Edelman, Dr. Jeff Bezanson and Stefan Karpinski. Both Shah and Bezanson were co-workers at *Interactive Supercomputing*, a company founded by Edelmann. Shah also went to graduate school with Karpinski. Through individual conversations they all shared their wish to have a better language for scientific computing. Hence, through a collective email thread the idea to invent their own programming language was born.[1].

The main goal of Julia is to "have machine performance without sacrificing human convenience".[2] While high-level languages like Python or R are simple to use, their performance often is suboptimal. On the other hand low-level languages like C or C++ achieve much better performance at the expense of readability. Hence, "users often prototype algorithms in a user-friendly language such as Python but then have to rewrite them in a faster language.[...]Julia circumvents that two-language problem because it runs like C, but reads like Python."[3].

2.1 Style

The syntax of Julia is reminiscent of both Python and Matlab as one can see in the following example of a simple Julia function which uses the sieve of Eratosthenes to find all primes up to a given number. The function gets an integer and finds every prime to *max* by iterating over all numbers from 2 to *max* and for every remaining number eliminating all multiples of this number. In the end all remaining numbers are prime.

```
function eratosthenes(max::Int64)
    # get an array of size max, every entry is a possible prime
    prim = fill(true, max)
    prim[1] = false
    for i in 2:isqrt(max)
        if prim[i]
            # mark every multiple of i that is not yet marked
            for j in i*i:i:max
                prim[j] = false
            end
        end
    end
    return prim
end
#take input from user
number = parse(Int64, Base.prompt("limit"))
prim = eratosthenes(number)
#print all indices of elements which are true
print(findall(prim))
```

The more a computer knows about the program, the better it is at executing it. This applies for example to variable types. Therefore, other programming languages that specialize in high-performance are typically statically typed. All types are defined and checked during compilation which guarantees excellent performance. However, describing types can take effort. Hence, Julia tries to compromise by allowing the user to write code dynamically which is still quite fast if it is written the "Julian way".[2]

To achieve this performance Julia also uses just-in-time compilation with LLVM. The code is compiled piece by piece shortly before its execution. The translation in machine code takes serveral steps. First the code is "lowered" into so-called "bytecode". This bytecode can be observed with `@code_lowered`. For example for a simple addition of two whole numbers one can see with `code_lowered` that this transformes to a basic addition on integers which returns another integer in the end.

```
@code_lowered 2+3
```

```
CodeInfo(
1 - %1 = Base.add_int(x, y)
|__      return %1
) => Int64
```

In this form each variable is assigned exactly once and loops are transformed to simple goto structures. Next the code is translated to "typed code", oberservable with `@code_typed`. Typed code is very similar to lowered code, but it optimizes the implementation of certain function based on type information. Hence, for each sub-function which is called the expected type returned is calculated in the example.

```
@code_typed 2+3
```

```
CodeInfo(
1 - %1 = Base.add_int(x, y)::Int64
|__      return %1
) => Int64
```

Next Julia creates LLVM IR (intermediate representation) which in this case is "an in-memory representation that is generated and consumed by LLVM libraries."[4] This IR can be observed with `@code_llvm`. This code already resembles an assembly representation.

```
@code_llvm 2+3
```

```
;  @ int.jl:87 within `+`
; Function Attrs: uwtable
define i64 @"julia_+_1901"(i64 signext %0, i64 signext %1) #0 {
top:
    %2 = add i64 %1, %0
    ret i64 %2
}
```

Finally, this IR is translated to native code which is just binary code in memory, but can be inspected with `@code_native` which displays it in assembly language.[4] Here, the addition is split into plain assembly calls.

```
@code_native 2+3
```

```
        .text
; ┌ @ int.jl:87 within `+`
        pushq   %rbp
        movq    %rsp, %rbp
        leaq    (%rcx,%rdx), %rax
        popq    %rbp
        retq
        nopw    (%rax,%rax)
; └
```

Aside from its compilation Julia is a multi-paradigm language, as it is object-oriented, but also functional and imperative. It is open-source, allows to call C functions directly and has support for unicode.[5] Furthermore, "Julia can also be embedded in other programming languages."[6]

2.2 Performance

The performance of Julia on simple functions can be observed in Figure 1.

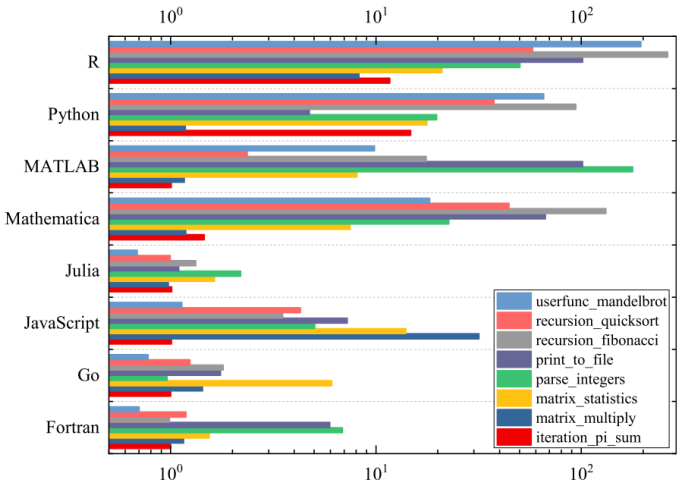


Figure 1: Benchmarks for simple functions[6]

The compiler performance of each language is tested "on a range of common code patterns, such as function calls, string parsing, sorting, numerical loops, random number generation, recursion, and array operations".[7] The functions are implemented to be comparable and not to achieve peak performance. For example the recursive fibonacci is written with a similar double recursion in each language. Then the performance is plotted normalized against the performance of a comparable C implementation.[7] One can see that the performance of naive implementations of high-level languages like Python or R is much worse than the correspondent C implementation. Meanwhile the naive Julia implementation achieves very similar results to the C implementation. The worst result arises for integer parsing, where the runtime is approximately twice as much. However, for the implementation of a simple Mandelbrot function the performance is even better than the performance of C. Julia also achieves high performance in parallel computing both on CPU and GPU which will be discussed later on. For further information on the reasons for Julia's perfomance I recommend [2].

3 Parallel Programming in Julia

Julia already has automatic parallel programming, which is used for example for matrix multiplication. Beyond that Julia features three main types of parallel programming which are interesting for HPC. It has OpenMP-like multi-threading contained in the `Threads` standard libary, a wrapper for the portable message passing system MPI and its own main implementation of message passing for distributed-memory systems contained in the `Distributed` module as part of the standard library.

3.1 Multithreading

"The main multithreading approach is to use the `Threads.@threads` macro which parallelizes a for loop to run with multiple threads"[8]. In the following the benchmark of a simple function adapted from [8] which gets an array A and calculates the array B where every entry is the square root of the corresponding entry of A can be observed.

```
function threaded_sqrt_array(A)
    B = similar(A)
    @threads for i in eachindex(A)
        @inbounds B[i] = sqrt(A[i])
    end
    B
end
```

The function `similar()` creates an array with the same type and dimensions of the given array. Normally Julia uses bound checking to ensure program safety when

accessing arrays. With the `@inbounds` command, this bound checking is skipped to improve performance. The function creates an array of dimension 1000 times 1000 filled with random float values and benchmark the performance on that array compared with a similar function which does not use multi-threading.

```
A=rand(1000, 1000)
@btime threaded_sqrt_array(A);
@btime sqrt_array(A);
```

where `sqrt_array` is the function without multi-threading. The `@btime` command is used to benchmark the execution time. For five seconds the function is executed as often as possible and in the end the minimal time is displayed. We get the following results:[1]

Number of threads	parallelized benchmark	serial benchmark
2	1,858ms	2.755ms
4	1.703ms	2.754ms
8	1.688ms	2.748ms

3.2 Multiprocessing

For multi-processing Julia's main approach is the package `Distributed`. Julia can be started with n processes on the command line with `julia -p n`. The `-p` command loads the package `Distributed` automatically. One can also specify the number of processes from inside Julia by using the `addprocs(n)` command. To call a function f on a specified process id one can use the command

```
remotecall(f, id::Integer, args...; kwargs...)
```

which evaluates to a `Future` object. A `Future` object is a placeholder for a single computation of unknown termination status and time. It is possible to fetch these objects by calling the `fetch` command. Normally, one does not specify process IDs explicitly, but the function `remotecall()` provides finer control. Another useful command for multi-processing is `@spawnat p expr`. This creates a closure on the expression *expr* and runs said closure on process p. Here, p can be the process ID, but it is also possible to use `:any` and let the compiler decide which process to use or to use `spawn` where the process ID is not specified at all. With these commands it is possible to create a simple program with two processes which creates an array of size two times two on process two and adds one to each coordinate.

```
r = remotecall(rand, 2, 2, 2)
s = @spawnat :any (1 .+ fetch(r))
fetch(s)
```

[1] tested on Intel(R) Core(TM) i5-8250U GPU @1.60 GHz

Even though the function did not specify on which process it wants to calculate the addition the compiler is smart enough to use the process which owns r.[2][9]

3.3 GPU offload

Julia implements different methods for GPU offload for different vendors. There is CUDA for NVIDIA, oneAPI for Intel and ROCm for AMD. CUDA "is mature, [and] has been under development since 2014"[10]. ROCm is still under development, although functional for simple usecases and oneAPI is still in early development. However, the effort for switching from one API to another typically only involves mechanical substitutions of individual API calls. Aside from the vendor-specific packages, Julia also offers
`KernelAbstractions.jl` (KA.jl), a package which supports CUDA, Intel and AMD. It focuses on performance portability and therefore also works on CPU, although due to its bias towards GPU programming, the performance might be suboptimal.[11] In general, for GPU computing, the vendor-specific solutions still offer higher performance as of now.

4 HPC in Julia

4.1 Benchmarks

In [12] the performance of Julia both with CPU and GPU on different HPC hardware was tested. I want to focus on the results on two CPU systems: Intel's Xeon Gold 6230 (Xeon) with a theoretical peak memory bandwidth of 281.6GB/s and AMD's EPYC 7742 (EPYC) with 409.6GB/s, as well as four GPU systems: NVIDIA's Tesla A100 (A100) with 2039 GB/s and Tesla V100 (V100) with 900 GB/s. and AMDs Instinct MI100 (MI100) with 1228GB/s and Instinct MI50 (MI50) with 1024GB/s.[3].

The theoretical peak memory bandwidth with the achieved peak memory bandwidth is computed using the McCalpin STREAM benchmark. This benchmark program uses five operations and measures the execution time for each operation. The operations are:

- Copy: $A[i] = B[i]$
- Mul: $A[i] = s \cdot B[i]$
- Add: $A[i] = B[i] + C[i]$
- Triad: $A[i] = B[i] + s \cdot C[i]$

[2]Note that it is important to put brackets around the expression even though this is not specified in the documentation

[3]More on the platform details can be found in the appendix in 7.1

- Dot: $R = R + (A[i] \cdot B[i])$

where s and R are scalars and A, B and C matrices. Then the bandwidth is derived, using the simple formula:

$$\text{“bandwidth”} = \frac{\text{total amount of data moved}}{\text{execution time}}.$$

This formula is based on a few assumptions on the architecture of computers. More on the derivation of the formula can be found in [13].

For CPU the performance of Julia's multithreading is compared with both OpenMP and Kokkos. Unfortunately the authors were not able to also compare the results with KA.jl on CPU due to “a bug with private memory use in for-loops when used in conjunction with `@synchronize`”.[7] In figure 2 one can see that Julia achieves very similar performance to OpenMP and Kokkos on both CPUs without any outlier.

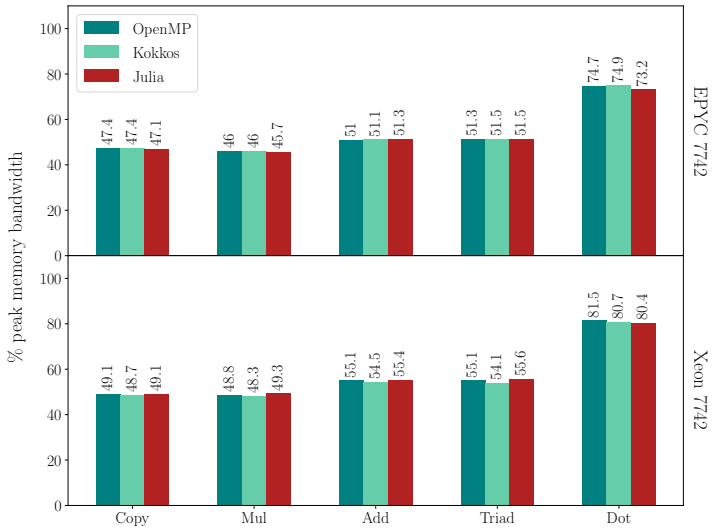


Figure 2: CPU results[12]

For GPU on NVIDIA the authors compared both CUDA.jl and KA.jl with CUDA, OpenCL and Kokkos. One can see in figure 3 that KA.jl has slightly worse performance than others as stated before. However, it still holds up. The performance of CUDA.jl essentially matches the performance of CUDA, beating the performance of Kokkos in copying, multiplication and scalar product calculations.

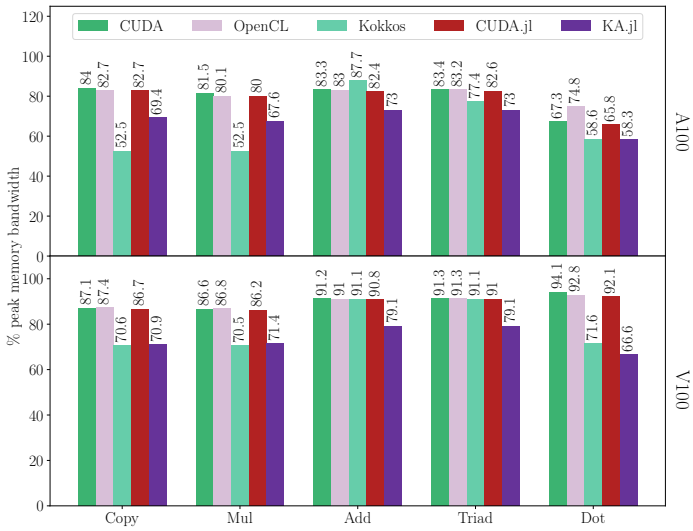


Figure 3: GPU results on NVIDIA[12]

For GPU on AMD the results on AMDGPU.jl which is a wrapper for ROCm and KA.jl are compared to OpenCL, HIP and Kokkos. In figure 4 one can see that OpenCL, HIP and Kokkos all achieve better performance than AMDGPU.jl and KA.jl regardless of the operation observed. However, the performance of AMDGPU.jl and KA.jl is very similar, "indicating KA.jl's API surface maps more directly to AMDGPU.jl"[12].

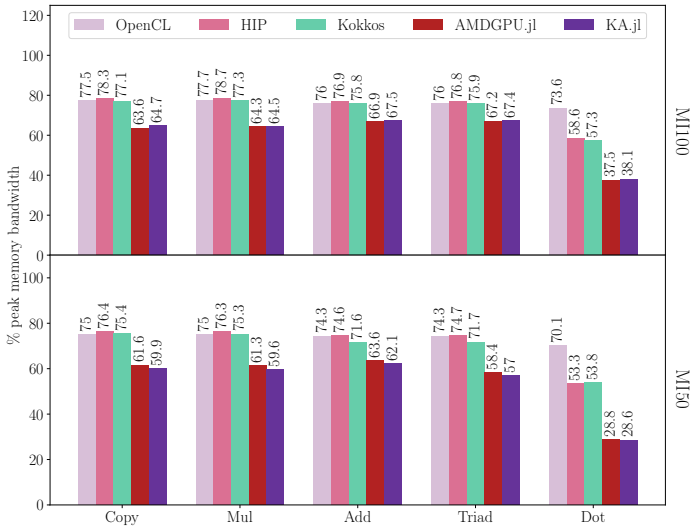


Figure 4: GPU results on AMD[12]

4.2 Projects

There already exist a few larger HPC projects which were written in Julia. Two of them will be presented in this section.

4.2.1 Oceananigans.jl

Oceananigans.jl was developed as part of the Climate Modelling Alliance project, a "coalition of scientists, engineers, and applied mathematicians from Caltech, MIT, and NASA's Jet Propulsion Laboratory."[14]. Their goal is to "provide the accurate and actionable scientific information needed to face the coming [climate] changes."[14] The package is suitable for research as well as students and first-time programmers. It provides "finite volume simulations of the nonhydrostatic and hydrostatic Boussinesq equations on CPUs and GPUs".[15] This approximation is "valid for weakly nonlinear and fairly long waves".[16]

4.2.2 Celeste.jl

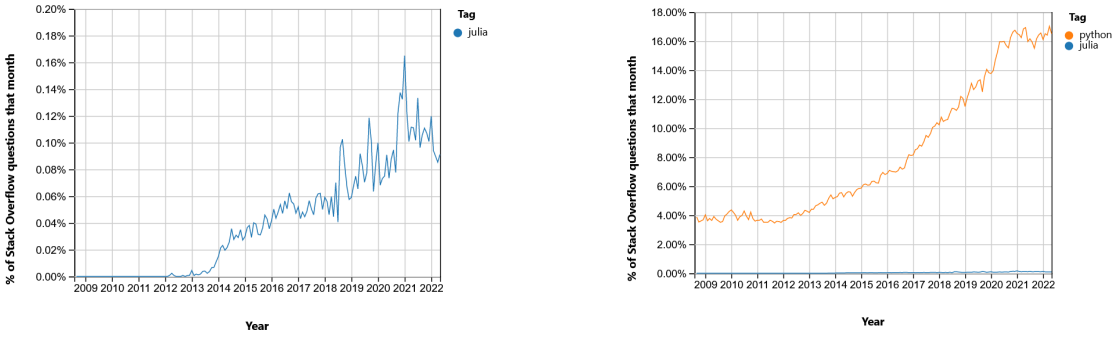
Another HPC project which is written in Julia is the Celeste project. In 2014 its development was started by a team of astronomers, physicists, computer engineers and statisticians. Their goal was to process the Sloan Digital Sky Survey dataset. This dataset was developed from the year 2000 onwards and provides telescope data of around 500 million stars and galaxies, around 35% of the sky. The Celeste project aimed to catalog all these stars and galaxies. They decided on this project not only because of its topic, but also because of the size of the data involved. To process all the data, consisting of 178 TB, the researchers used 1.3 million threads on 9,300 Knights Landing (KNL) nodes of the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC). They decided to use Julia to test its claim to combine both productivity and performance. In the end they produced the most accurate catalog of 188 million astronomical objects in just 14.6 minutes while achieving peak performance of 1.54 petaflops.[17] Up until then only a few languages such as C, C++ and Fortran have executed an application which exceeds one petaflop.[18]

5 State of the art

5.1 Popularity

Julia's popularity has grown a lot since its release in 2012. One indicator for that is the number of times a question asked on Stack Overflow contained the tag "Julia". In Figure 5a one can see that the number of question had an upwards trend up until 2021. Since then one can see a small drop. It would be interesting to observe in the future whether this indicates a stagnation or even downwards trend or not.

However, by comparing the number of questions tagged "Julia" with those tagged "Python" one can see in 5b that this increase is irrelevant if compared to Python's popularity. "There are entire Python packages like TensorFlow or PyTorch that have much more traction than Julia as a language"[19]. Of course this comparison is hardly fair, as Python is much older and one of the most popular programming languages. Nevertheless it helps contextualize these grow statistics.

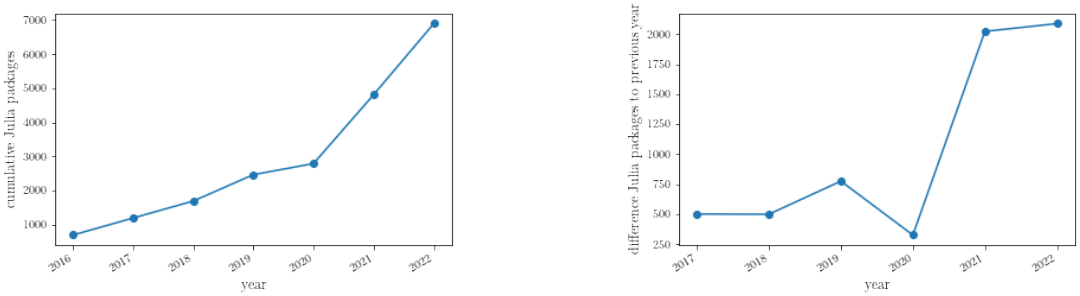


(a) Julia

(b) Julia and Python

Figure 5: Stack Overflow trends [20]

The number of packages in Julia indicate similar trends. In Figure 6a one can see the cumulative number of packages that is ever-increasing. In Figure 6b one can see the increase of packages compared to the previous year. Since 2020 the velocity of new packages developing is much higher. However, the comparison with Python again shows how small these numbers are. As of January 2020, Python has about 350,000 packages, more than 50 times the amount of Julia packages.[21] However, since Julia is not as mature as Python it is still possible to steer its direction by contributing which is easier than in other languages, because all packages are written in Julia.[19]



(a) Cumulative

(b) Compared to previous year

Figure 6: Number of packages of Julia [22]

5.2 Downsides

One of the greatest downside mentioned a lot is the slow compile time and the slow time to first plot. When writing code in any IDE you can "see a *noticable* lag

before the text appears."[23] This problem occurs especially often when plotting is involved. However, since the release of Julia this latency is improving. Nonetheless, this problem will always exist in Julia and creates use cases where it is not feasible to use Julia. Another mayor downside of Julia is its large memory consumption.[23] For the simple function

```
function print_hello_world()
    println("Hello World!")
end
```

and track its memory allocations with

```
@time print_hello_world();
```

a comparison of memory allocation and execution time yields that the first execution allocates 1.024 MiB with 18,56k allocations in 0.030429 seconds. The next execution only allocates 288 Bytes with 11 allocations in 0.000292. This also shows that to compare execution time and memory allocations it is necessary in Julia to execute a function more than once.

However, the greatest downside is still the immaturity of Julia. As mentioned before there are very few packages in Julia and while using packages you can often encounter "bug[s], redesigns, or performance problems with libraries".[24] Also, up until now there are few popular libraries in Julia. "In Python, everybody knows, for example, to use pandas when working with dataframes [...] In Julia, it's not too rare to want a functionality and find three packages that do it in slightly different ways, all of them immature and light on features."[23][4]

6 Conclusion

In conclusion Julia succeeds in what it tries to accomplish. Probably it will never be able to replace any popular language such as Python or C. If one wants to focus solely on performance it is still easier to use C. However, if one does not need the performance and wants to focus on productivity, it is still easier to use Python. Nevertheless, especially for HPC the combination of efficiency and performance, mainly on CPUs and NVIDA GPUs could be very beneficial. Since the package `AMDGPU.jl` is still very immature the performance on AMD GPUs will hopefully be improved in the future. Projects like `Celeste.jl` show that it is already feasible to use Julia for HPC projects.

[4]For personal experiences on the downsides of working with Julia I recommend [23] and [24]

7 Appendix

Table 7.1: Platform details

Vendor	Name	Architecture	Abbreviation	Device Type	Theoretical Peak Mem. Bandwidth (GB/s)	Theoretical Peak FP32 FLOP/s (TFLOP/s)
Intel	Xeon Gold 6230	Cascade Lake	Xeon	HPC CPU (20C*2, 2S)	281.6	4.096
AMD	EPYC 7742	Zen2 (Rome)	EPYC	HPC CPU (64C*2, 2S)	409.6	9.216
NVIDIA	Tesla A100 (SXM 80 GB)	Ampere	A100	HPC GPU	2039	19.490
NVIDIA	Tesla V100 (PCIe 16GB)	Volta	V100	HPC GPU	900	14.130
AMD	Instinct MI100	CDNA	MI100	HPC GPU	1228	23100
AMD	Instinct MI50	GCN (Vega 20 GL)	MI50	HPC GPU	1024	13300

Bibliography

[1] Interview with julia language co-founders. https://www.youtube.com/watch?v=VgZm53qgj9Q. Accessed: 2022-08-05.

[2] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fresh approach to numerical computing. *SIAM Review, Volume 59*, 2017.

[3] Jeffrey M. Perkel. Julia: come for the syntax, stay for the speed. https://www.nature.com/articles/d41586-019-02310-3. Accessed: 2022-08-08.

[4] What is the difference between @code_native, @code_typed and @code_llvm in julia? https://stackoverflow.com/questions/43453944/what-is-the-difference-between-code-native-code-typed-\and-code-llvm-in-julia/43456211#43456211. Accessed: 2022-09-12.

[5] Julia 1.7 documentation. https://docs.julialang.org/en/v1/. Accessed: 2022-08-09.

[6] Kaifeng Gao, Gang Mei, Francesco Piccialli, Salvatore Cuomo, Jingzhi Tu, and Zenan Huo. Julia language in machine learning: Algorithms, applications, and open issues. *Computer Science Review, Volume 37*, 2020.

[7] Julia micro-benchmarks. https://julialang.org/benchmarks/. Acessed: 2022-08-10.

[8] Parallelization. https://enccs.github.io/Julia-for-HPC/parallelization/. Accessed: 2022-08-10.

[9] Multi-processing and distributed computing. https://docs.julialang.org/en/v1/manual/distributed-computing/l. Accessed: 2022-08-11.

[10] Juliagpu. https://juliagpu.org/. Accessed: 2022-08-11.

[11] Kernelabstractions. https://juliagpu.gitlab.io/KernelAbstractions.jl/. Accessed: 2022-08-11.

[12] Wei-Chen Lin and Simon McIntosh-Smith. Comparing julia to performance portable parallel programming models for hpc. pages 94–105, 2021.

[13] The stream benchmark. https://stackoverflow.com/questions/56086993/what-does-stream-memory-bandwidth-\benchmark-really-measure. Accessed: 2022-08-15.

[14] Climate modeling alliance. https://clima.caltech.edu/. Accessed: 2022-08-17.

[15] https://github.com/clima/oceananigans.jl. https://github.com/CliMA/Oceananigans.jl. Accessed: 2022-08-18.

[16] Boussinesq approximation (water waves). https://en.wikipedia.org/wiki/Boussinesq_approximation_(water_waves). Accessed: 2022-08-18.

[17] Parallel supercomputing for astronomy. https://juliacomputing.com/case-studies/celeste/. Accessed: 2022-08-18.

[18] What julia aims to accomplish? https://juliadatascience.io/julia_accomplish. Accessed: 2022-08-18.

[19] The rise of the julia - is it worth learning in 2022? https://www.datacamp.com/blog/the-rise-of-julia-is-it-worth-learning-in-2022. Accessed: 2022-08-22.

[20] Stack overflow trends. https://insights.stackoverflow.com/trends. Accessed: 2022-08-22.

[21] all_packages. https://pypi.org/project/all-packages/. Accessed: 2022-08-22.

[22] Newsletter january 2022 - julia growth statistics. https://juliacomputing.com/blog/2022/01/newsletter-january/. Accessed: 2022-08-22.

[23] What's bad about julia? https://viralinstruction.com/posts/badjulia/. Accessed: 2022-08-23.

[24] What don't you like about julia for "serious work"? https://discourse.julialang.org/t/what-dont-you-like-about-julia-for-serious-work/54591. Accessed: 2022-08-23.