Introduction to the Julia Language

Xuanyu Chen, Tony Xiao, Justin Jia

December 12, 2017

1 History

In 2009, Jeff Bezanson, Stefan Karpinski, Viral B. Shah and Alan Edelman started their work on Julia. On February 14th 2012, they published a blog post to explain the mission of Julia language. The Julia community has grown rapidly since then. Until September 2017, Julia has been downloaded for more than 1.2 million times.

Julia's initial version number is 0.1.2 (released in early 2012). Version 0.2 was launched on November 2013. Currently, releases earlier than 0.5 are deprecated and no longer maintained. Only bug fixes releases will be published for Julia 0.5. The latest stable version 0.6.1. [1]

2 Motivation

"We are greedy." Jeff Bezanson, Stefan Karpinski, Viral Shah and Alan Edelman stated on the blog. They wanted to have a programming language that are powerful on many different areas, including scientific computing, machine learning, data mining, large-scale linear algebra, distributed and parallel computing.

They wanted the speed of C along with the dynamism of Ruby. They wanted a programming language that is homoiconic, with true macros like Lisp, but with obvious and familiar mathematical notation like Matlab. "They wanted something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, and as good at gluing programs together as the shell." [2]

"They also wanted a language syntax to be clean and simple. For example, programmers should not need to explicitly write down variable types. For computing strategies, they wanted a language to provide the distributed power of Hadoop without the layers of impenetrable complexity." [2]

When they finally integrated all they wanted together, Julia was introduced to the world.[3]

3 Features

According to the Julia official website [4]:

- Multiple dispatch: providing ability to define function behavior across many combinations of argument types
- Dynamic type system: types for documentation, optimization, and dispatch
- Good performance, approaching that of statically-typed languages like C
- A built-in package manager
- Lisp-like macros and other meta-programming facilities

- Call Python functions: use the PyCall package
- Call C functions directly: no wrappers or special APIs
- Powerful shell-like abilities to manage other processes
- Designed for parallel and distributed computing
- Coroutines: lightweight green threading
- User-defined types are as fast and compact as built-ins
- Automatic generation of efficient, specialized code for different argument types
- Elegant and extensible conversions and promotions for numeric and other types
- Efficient support for Unicode, including but not limited to UTF-8

4 Syntax

4.1 Code

```
function helloWorld(num)
    localnum = num
    for n = 1:localnum
        if n == 1
            println("This program Will print $localnum times hello world")
            println("hello world, $(localnum - 1) times left")
            println("$tmp")
        else
            remains = localnum - n
            println("hello world, $remains times left")
        end
    end
end
```

4.2 Analysis

1. Similar to R and Python, Julia doesn't require programmers to write code that explicitly specifies types in most cases.
2. The flow control in Julia is similar to Python and R. However, it uses "begin" and "end" as scoping delimiters, while R uses and Python uses indentation.

5 What makes Julia Fast

Unlike other dynamic languages (i.e. R and Python) that are designed on "two-tiered architectures", which is using system level languages (i.e. C) to implement performance critical parts, Julia is a really fast language by itself. It is carefully designed to only keep important dynamic features while omitting others that could impede performance.
Some key features include:

- Rich type information

- Code specialization against runtime types
- JIT compilation using LLVM without configuration files
- Variable type cannot be changed during its lifetime
- Types are immutable

Many features seem like limitations at first, but these limitations actually played an important role in keeping Julia fast. For example, "variable type cannot be changed during its lifetime" rule enables many optimizations that cannot be done in Python and R. These features allow Julia to maintain its dynamic nature while keeping its performance close to the C level.

Because Julia itself is really fast, many of its core language functions are implemented in Julia. Without "two-tiered architectures", Julia can benefit from whole program optimization. Because different languages have different APIs and ABIs, it is hard to optimize them as a whole. Interfaces for communication between two different programming languages in "two-tiered architectures" will also inevitably cause overhead.[5]

Julia has attracted some high-profile clients, from investment manager BlackRock, who uses it for time-series analytics, to the British insurer Aviva, who uses it for risk calculations. The language was released under MIT license, which means that Julia can be used and/or modified without any right infringement. Although Julia creators also founded Julia Computing that provides paid support, training and consulting services to users, Julia language itself remains free and open source.

At the 2017 JuliaCon conference, Jeff and others announced that the Celeste project "implementation is written entirely in Julia and utilizes high-level constructs for shared and distributed memory parallelism. The project utilizes the Knights Landing based NERSC Cori Phase II supercomputer – the fifth most powerful machine in the world – making it one of the largest generative models ever developed."[6]

6 Parallel Computation Facilities

Julia provides parallel computation facilities based on message passing design. However, its implementation is different from MPI's. When programming in Julia, programmers usually only want to manage one process explicitly and let it control other processes.

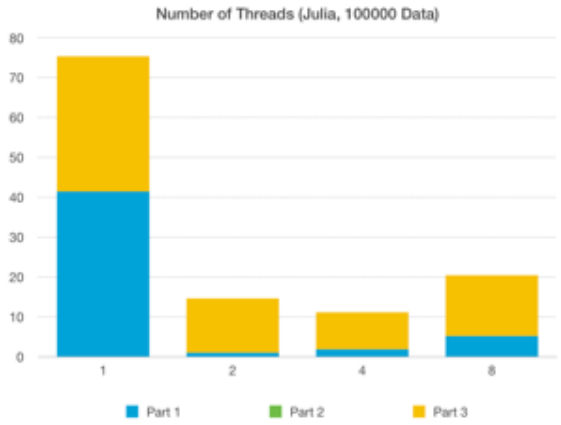
The syntax of Julia's parallel library is similar to R's parallel package (also known as snow). APIs are provided to users through functions, unlike OpenMP which uses pragmas.

Like OpenMP and other libraries, shared memory design is also supported in Julia through SharedArray type. According to the official guide, "Shared Arrays use system shared memory to map the same array across many processes."

Julia provides Remote Reference and Remote Call APIs. Remote Reference allows processes to reference objects that are managed by other processes. Remote Call allows processes to call functions to run on other processes and receive results as Future type objects.

7 Timing Comparison

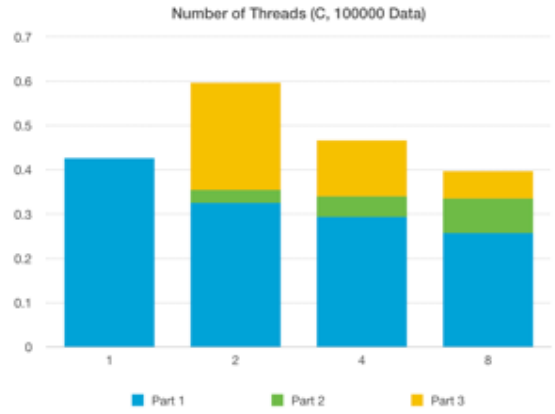
7.1 Graphs



Number of Threads (Julia, 100000 Data)

NUMBER OF THREADS	1	2	4	8
Part 1	41.463	1.028	1.855	5.208
Part 2	0	0.005	0.005	0.007
Part 3	33.972	13.661	9.402	15.298

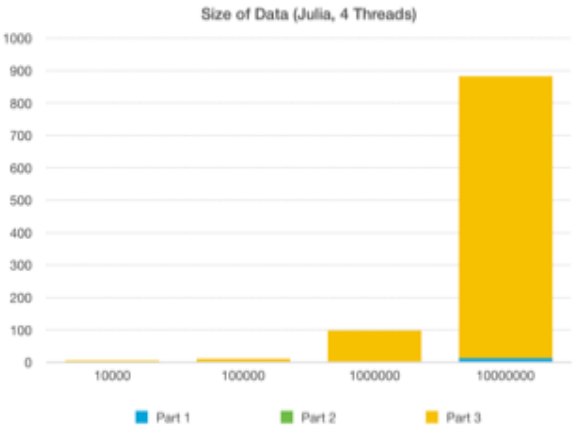
Figure 1: Julia Thread Comparison



Number of Threads (C, 100000 Data)

NUMBER OF THREADS	1	2	4	8
Part 1	0.426	0.325	0.294	0.257
Part 2	0	0.030	0.046	0.078
Part 3	0	0.241	0.126	0.062

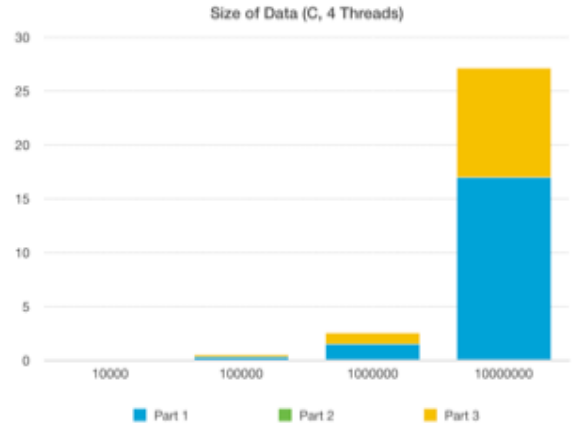
Figure 2: C Thread Comparison



Size of Data (Julia, 4 Threads)

SIZE OF DATA	10000	100000	1000000	10000000
Part 1	2.581	1.855	2.599	12.293
Part 2	0.005	0.005	0.006	0.05
Part 3	3.735	9.402	94.946	870.638

Figure 3: Julia Data Comparison



Size of Data (C, 4 Threads)

SIZE OF DATA	10000	100000	1000000	10000000
Part 1	0.06	0.294	1.47	16.973
Part 2	0.042	0.046	0.041	0.046
Part 3	0.012	0.126	1.028	10.087

Figure 4: C Data Comparison

7.2 Analysis

Graphs above are Timing Comparisons between Julia and C on a 4 Cores Machine.

We divided the program running time into three parts (both C and Julia). In part 1, each thread calculates the prefix sum in its own range and stores the cumulative sum, which is the last element in its sub-array, into a shared array. In part 2, only one thread calculates the prefix sum of the shared array. In part 3, each thread adds the value in the shared array accordingly to its thread number to its own sub-array.

As we can see from Figure 1, code written in Julia gets better performance as the number of threads increases until it reaches the total number of cores of the computer (4 cores). The time spent on part 1 and part 3 decreased because: as the number of threads increases, the workload of each thread decreases. Part 2's workload is only related to the total number of threads. However, since it is really small it can be ignored. Because the machine only has 4 cores, extra context switch overhead caused the code to perform worse on 8 threads than 4 threads.

C program has similar performance characteristic. However, some work done in Julia version's part 1 was moved to C version's part 2. C version assigned value to the shared array in part 2 instead of part 1. Therefore, part 2 in C took longer time. Because C is really fast, the sample size is not large enough to show the time difference between single thread version and multi-thread version. Overall, as we can see, Code written in Julia is slower (but not much) to code written in C.

In Figure 3 and 4, performance was measured using different data sizes. Since we keep the number of threads a constant value, part 2 in all cases took constant time.

As data size increases, rates of time increase are the same in both Julia and C. Julia's DArray is relatively efficient. However, the implementation of Julia's SharedArray is very slow, and it caused the huge total time difference we can seen above.

8 Author Contributions

Justin: Analyze data, graph, why Julia fast and application, some Julia coding and worked on the report.
Tony: Analyze data, coding, why Julia fast and application, time comparison simulation, and worked on the report.
Xuanyu: Finding resources, analyze data, history and motivation of Julia, and worked on the report.

A Parallel Prefix Sum in C

```
void seqprfsum(int *u,int m)
{  int i,s=u[0];
   for (i = 1; i < m; i++) {
      u[i] += s;
      s = u[i];
   }
}

void parprfsum(int *x, int n, int *z)
{
   #pragma omp parallel
   {  int i,j,me = omp_get_thread_num(),
          nth = omp_get_num_threads(),
          chunksize = n / nth,
          start = me * chunksize;

      // Part 1
      seqprfsum(&x[start],chunksize);
      #pragma omp barrier

      // Part 2
      #pragma omp single
      {
      for (i = 0; i < nth-1; i++)
         z[i] = x[(i+1)*chunksize - 1];
      seqprfsum(z,nth-1);
      }

      // Part 3
      if (me > 0) {
         for (j = start; j < start + chunksize; j++) {
            x[j] += z[me - 1];
         }
      }
   }
}
```

B Parallel Prefix Sum in Julia

```
# Require DistributedArrays package
# pkg.add("DistributedArrays")
# Julia index starts from 1

function seqprfsum(u)
    m = length(u)
    s = u[1]
    for i = 2:m
        u[i] += s
        s = u[i]
```

```
    end
end

function parprfsum(x)
    x = distribute(x) # Distribute x (Array) to all processes
    ps = procs(x) # Find all processes that contain x (DArray)
    # Allocate z (SharedArray) that is shared across processes
    z = SharedArray{Int64,1}(length(ps) + 1)

    # Part 1
    # Wait for all the processes finished (Act as an barrier)
    @sync for p in ps
        @spawnat p begin # run asynchronously on process p
            me = myid()
            # Find local parts of x (DArray) that is stored on the current process
            locx = localpart(x)
            seqprfsum(locx)
            z[me] = locx[length(locx)]
        end
    end

    # Part 2
    seqprfsum(z)

    # Part 3
    # Wait for all the process finished (Act as an barrier)
    @sync for p in ps
        @spawnat p begin
            me = myid()
            # Find local parts of x (DArray) that is stored on the current process
            locx = localpart(x)
            if me > 2
                for i in 1:length(locx)
                    locx[i] += z[me - 1]
                end
            end
        end
    end
    x
end
```

References

[1] *"Julia | Definition, Programming, History."* Cleverism, Cleverism, https://www.cleverism.com/skills-and-tools/julia/

[2] Why We Created Julia *julialang.org.* https://julialang.org/blog/2012/02/why-we-created-julia

[3] "Julia (Programming Language).", *Wikipedia, Wikimedia Foundation.* 11 Dec. 2017, en.wikipedia.org/wiki/Julia

[4] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, et al. *JuliaLang.* https://docs.julialang.org/en/stable/index.html

[5] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman *Julia: A Fast Dynamic Language for Technical Computing* https://arxiv.org/pdf/1209.5145v1.pdf

[6] JuliaCon 2017 *JuliaCon.org.* http://juliacon.org/2017/