Vojtech Nikl

The Julia Language

Survey

March 2, 2016

Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic

Abstract

Dynamic languages, which have become very popular for scientific computing, are generally considered highly productive, but lacking in performance. This survey presents Julia, a new dynamic language designed specifically for technical computing, which is developed with an emphasis on performance and compact and easy-to-understand source code, which allows most of the Julia's libraries to be written in Julia itself, while also being able to incorporate C and Fortran libraries.

The content of this survey is intended for programmers, who come from the area of performance computing, as well as others who are searching for a new fast dynamic language and want to find out the possibilities that this language offers in a compressed format. It is based on the official documentation, related papers and personal experience and focuses on the performance aspects of Julia, parallel programming, profiling and code optimizations.

Contents

1 Introduction

High level environments such as Matlab, Octave and R, provide greatly increased convinience and productivity. However, C and Fortran remain the gold standard for computationally-intensive tasks. As a result, the most challenging areas of technical computing have benefited the least from the increased abstraction and productivity offered by higher level languages.

The two-tiered architectures have emerged as a compromise between inconvinience and performance. Programmers use dynamic languages to express high-level logic, while the heavy lifting is done in static laguages like C and Fortran. While this approach can be optimal for some applications, there are a few drawbacks. For some algorithms, especially the parallel ones, the code complexity can increase dramatically. To achieve high performance, one of many techniques used is vectorization, which is unnatural for many problems and migh generate large temporary objects which could be avoided with explicit loops. It would be preferable to write compute-intensive code in one productive language.

An alternative to the two-tier compromise is to enhance the performance of existing dynamic laguages. Projects like Python compiler framework PyPy[3] have been fairly succesful. However the design decisions made under the assumption that the language is interpreted tend to negatively impact the ability to generate efficient code.

Julia[1] is designed from the ground to take advantage of modern techniques for executing dynamic languages efficiently. As a result, Julia has the performance of a statically compiled language while providing interactive dynamic behaviour and productivity like Python, LISP or Ruby. This is achieved thanks to the Julia's core being based on the LLVM implementation.

[1] http://www.julialang.org/

2

Language Performance Comparison

In [1], the performance of Julia on a set of tasks was compared against other dynamic and static languages, including C, Matlab, Python and others (see Figure 2.1).

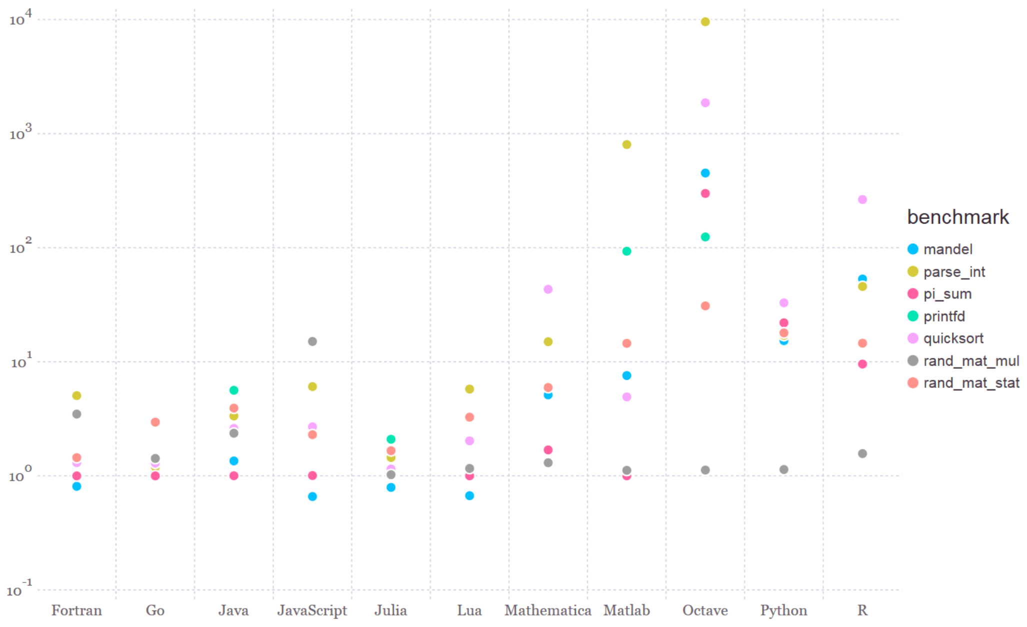

Fig. 2.1: Performance comparison of various language performing simple micro-benchmarks. Benchmark execution time relative to C. (Smaller is better, C performance = 1.0) [1].

The results show that Julia's performance is ahead of not only all other dynamic languages, but is even comparable to static languages. The results were obtained on a single core (serial execution) Intel(R) Xeon(R) CPU E7-8850 2.00GHz CPU with 1TB of 1067MHz DDR3 RAM, running Linux.

Similarly, the performance of different dynamic languages was compared in [2], this time on Apple MacBook (see Fig. 2.1). Again, Julia comes out on top.

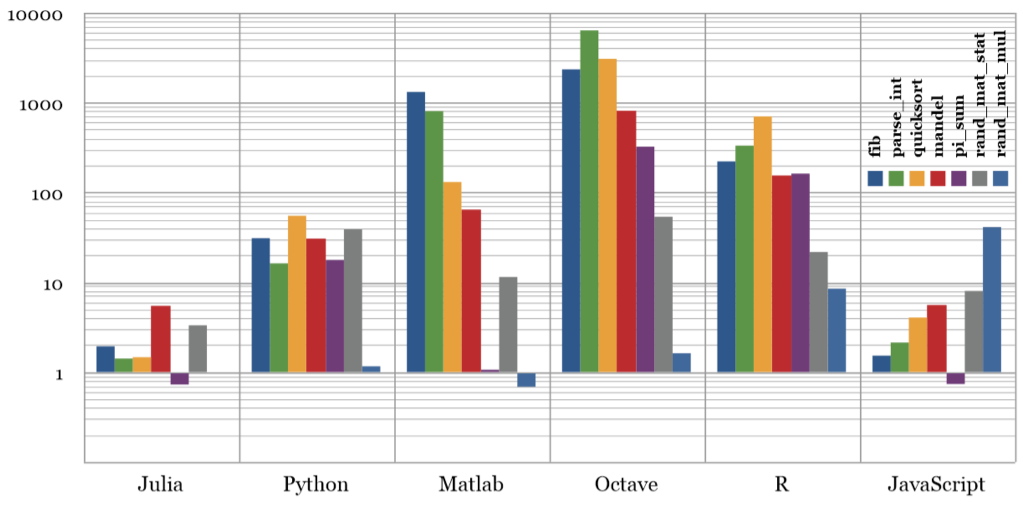

Fig. 2.2: Microbenchmark results (times relative to C++, log-scale). The C++ baseline was compiled by GCC 4.2.1, taking best timing from all optimization levels. Native implementations of array operations, matrix multiplication, sorting, are used where available. (Smaller is better, C++ performance = 1.0) [2].

3

Basic Syntax and Semantics Overview

This chapter will briefly describe Julia's syntax as a quick introduction to its basic programming rules and concepts. Note that Julia is a case-sensitive language. See online documentation for further details[1].

3.1 Variables

Variable names must begin with a letter (A–Z or a–z), underscore, or a subset of Unicode characters greater than ooAo. The variable syntax does not differ much from other dynamic languages:

```
x = 10
x = x + 1.0
x = "Hello World!"
```

Julia provides some predefined constants and functions, which can be overwritten:

```
pi            julia> π = 3.1415926535897...
pi = 3        julia> WARNING: imported binding for pi overwritten in Main
sqrt(100)     julia> 10
sqrt = 4      julia> WARNING: imported binding for sqrt overwritten in Main
```

However built in statements cannot be used as variables:

```
else = false    julia> ERROR: syntax: unexpected "else"
try = "No"      julia> ERROR: syntax: unexpected "="
```

3.2 Integers and Floating-point numbers

Basic types are shown in Table 3.1.

64-bit floating-point literals have common syntax:

[1] http://docs.julialang.org/en/release-0.4/manual/variables/

Bool	**Integer**		**Float**
	signed	unsigned	
`Bool`	`Int8`	`UInt8`	
	`Int16`	`UInt16`	`Float16`
	`Int32`	`UInt32`	`Float32`
	`Int64`	`UInt64`	`Float64`
	`Int128`	`UInt128`	

Table 3.1: Basic variable types.

```
Float64 :  1.0  1.  .5  -1.23  1e10  2.5e-4
```

The 32-bit version has `f` instead of `e`:

```
Float32 :  .5f0  2.5f-4
```

Half-precision floating-point numbers are also supported (`Float16`), but only as a storage format. In calculations they'll be converted to `Float32`.

The underscore _ can be used as digit separator:

```
0_000, 0.000_000_005, 0xdead_beef, 0b1011_0010
```

Special floating point values include

```
Inf16 Inf32 Inf -Inf16 -Inf32 -Inf NaN16 NaN32 NaN
```

Julia also supports Arbitrary Precision Arithmetics with `BigInt` and `BigFloat` types:

```
x = factorial(BigInt(40))  julia> 815915283247897734345611269596115894272000000
```

Binary, octal and hexadecimal literals are also supported:

```
0b10 0o7 0x16
```

Numeric literal coefficients are also supported, precedence is the same as unary operators:

```
2x^2  2^2x
```

Overflow behaviour is the same as in the C language.

```
x = typemax(Int64)  julia> 9223372036854775807
x = x + 1           julia> -9223372036854775808
```

3.3 Operators

All operators are shown in Table 3.2.

unary	`+ -`
binary	`+ - * /` \ (inverse division) `^` (power) `%` (modulo)
negation	`!`
bitwise	∼ (not) & (and) | (or) $ (xor) `>>>` (logical shift right) `>> <<` (arithmetic shift)
updating	`+= -= *= /=` \= ÷= `%= ^= &=` |= `$= >>>= >>= <<=`
comparison	`== != < <= > >=`

Table 3.2: Arithmetic operators

3.4 Complex and Rational Numbers

The imaginary part of complex numbers is written as *im*, which represents a global constant and is bound to the complex number i, representing the principal square root of -1.

$1 + 2im(-3 + 2im)/(1 - 2im)^2$

Julia has a rational number type to represent exact ratios of integers. Rationals are constructed using the // operator:

$6//9$

3.5 Characters and Strings

Characters are represented as a 32-bit bitstype and are interpreted as *Unicode code point.*

```
'A' 'a' '\u2200' '\x7f' '\177' '\t' '\n' '\0'
```

Strings are finite sequences of these characters and are delimited by double quotes. Julia also supports character and substring extraction in a number of different ways:

```
str = "Hello, world.\n"
str[1]                          julia> 'H'
str[end-1]                      julia> '.'
str[4:9]                        julia> 'lo, wo'
str[0]                          julia> ERROR: BoundsError()
```

An arbitrary number of strings can be concatenated together:

```
greet = "Hello"
whom = "world"
string(greet, ", ", whom, ".\n")        julia> "Hello, world.\n"
```

A special type of strings are *tripple quoted strings*, which are used for creating well-arranged long strings:

```
str = """
           Hello,
           world.
         """                 julia> "Hello, world.\n"
```

Common string operations are also supported, comparison proceeds lexicographically:

```
"abracadabra" < "xylophone"          julia> true
"abracadabra" == "xylophone"         julia> false
"abracadabra" != "xylophone"         julia> true
search("xylophone",'x')              julia> 1
contains("xylophone",'o')            julia> true
```

Julia has Perl-compatible regular expressions (regexes), as provided by the PCRE library, and are represented as strings prefixed with `r`.

```
r"^\s*(?:#|$)"                                julia> r"^\s*(?:#|$)"
ismatch(r"^s*(?:#|$)", "not a comment")       julia> false
match(r"^\s*(?:#|$)", "# a comment")          julia> RegexMatch("#")
```

3.6 Functions

In Julia, a function is an object that maps a tuple of argument values to a return value. The basic definition syntax is:

```
function f(x,y)
  x + y
end
```

Function bodies with a single (or compound) expression can be defined in a more compact form as:

```
f(x,y) = x + y
```

A function is called using the traditional parenthesis syntax:

```
f(2,3)      julia> 5
```

Arguments of the function are not copied during the call, they act as a new variable "binding". This behaviour is similar to other dynamic languages and *references* in C++.

The value returned by a function is the value of the last expression evaluated (by default the last expression of the body). The `return` keyword causes a function to return immediately.

```
function g(x,y)
  return x * y
  x + y
end
g(2,3)               julia> 6
```

Operators are functions and can be expressed using infix form:

```
1+2+3       julia> 6
+(1,2,3)    julia> 6
```

Functions can be defined without a name (*anonymous functions*):

```
x -> x^2 + 2x - 1       julia> (anonymous function)
```

The primary use for anonymous functions is passing them to functions which take other functions as arguments. A classic example is `map()`, which applies a function to each value of an array and returns a new array containing the resulting values:

```
map(x -> x^2 + 2x - 1, [1,3,-1])       julia> 3-element Array{Int64,1}: 2 14 -2
```

Functions can return multiple values using a tuple, which can be defined without parentheses.

```
function f(a,b)
  a+b, a*b
end;
x, y = f(2,3);    # x == 5, y == 6
```

It is possible to write functions that take arbitrary number of arguments (*varargs functions*). An ellipsis is following the position of the last argument. In this case, the `x` argument is bound to an iterable collection of zero or more values passed to the function:

```
bar(a,b,x...) = (a,b,x)
bar(1,2)             julia> (1,2,())
bar(1,2,3)           julia> (1,2,(3))
bar(1,2,3,4,5,6)     julia> (1,2,(3,4,5,6))
```

Functions can have optional arguments:

```
function multiply(x, y, z=10)
    x * y * z
end
multiply(2,3)            julia> 60
multiply(2,3,4)          julia> 24
```

3.7 Control Flow

Conditional `if-elseif-else` evaluation, where `elseif` and `else` are optional, is written as follows:

```
if x < y
  println("x is less than y")
elseif x > y
  println("x is greater than y")
else
  println("x is equal to y")
end
```

Julia also provides the so-called ”ternany operator”:

```
println(x < y ? "less than" : "not less than")
```

Conditional statements using && and || are also supported:

```
n >= 0 || error("n must be non-negative")
n == 0 && error("n cannot be zero")
```

Julia implements the two main types of loops, `while` and `for`:

```
while i <= 5           for i = 1:5
  println(i)              println(i)
  i += 1                 end
end
```

4

Code Profiling

The `Profile` module, when running, takes measurements of the running code and produces output to better understand how much time is spent in different parts of the code. The main usage is to identify application's "bottlenecks".

Profiling in Julia is known as "sampling" or statistical profiler. It periodically takes a backtrace during the runtime of the code and snapshots the current state of execution by capturing currently-running function and a line number. The more time is spent executing an individual line of code, the more frequent its record is in the backtraces. The backtraces occur at intervals of around a few milliseconds, so the collected data is subject to statistical noise.

Despite these limitation, sampling profilers have substantial strengths. No modifications to the code are necessary, it can profile Julia's core and optionally C and Fortran libraries, and depending on the tracking intervals, the performance overhead is very little.

To run the profiler, we have to specify a function to be profiled:

```
@profile myfunc()
```

If we want to profile the whole application, the main function has to specified. The profiling result, after running the `Profile.print()` command, looks similar to this:

```
Count File        Function                        Line
 3121 client.jl   _start                           373
 3121 client.jl   eval_user_input                   91
 3121 client.jl   run_repl                         166
  842 dSFMT.jl    dsfmt_gv_fill_array_close_open!  128
  848 none        myfunc                            52
 1510 none        myfunc                            35
```

Each line represents a particular spot in the code. The first column represents the amount of backtraces hitting this particular line of code (column 4) of the function printed in column 3. Statistically, the expect uncertainty for `N` samples collected on a line is on the order of $\sqrt{N}$. The one major exception is the garbage collector, which is written in C, but runs infrequently and is often very expensive.

5

Performance Hints and Enhancements

The following sections will briefly go through a few tips, patterns and techniques to further enhance Julia's performance.

Array bound checking Use `@inbounds` to eliminate array bounds checking within expressions:

```
@inbounds s += x[i]*y[i]
```

Performance annotations Use `@fastmath` to allow floating point optimizations (however note that it may slightly change the numerical results). Write `@simd` in front of an innermost for loop to enable vectorization (however this feature is still experimental and forcing vectorization incorrectly, for example with iteration-dependent loops, may lead to incorrect results).

```
@fastmath @simd for i=1:length(x)
    @inbounds s += sin(2pi*x[i]*y[i])
```

Subnormal numbers Treat subnormal numbers as zeros. Subnormal number have all exponent bits equal to zero and mantisa is non-zero. A call `set_zero_subnormals(true)` grants this permission for all floating point operations.

Deprecated functions Do not use deprecated functions, because they internally perform a lookup in order to print relevant warnings, which can cause significant slowdowns.

Parallel functions Use `@async` for parallel function execution instead of `@spawnat`. This code, which performs a single network round-trip to every worker:

```
responses = cell(nworkers())
@sync begin
    for (idx, pid) in enumerate(workers())
        @async responses[idx] = remotecall_fetch(pid, foo, args...)
    end
end
```

is faster than this code, which results in two network calls:

```
refs = cell(nworkers())
for (idx, pid) in enumerate(workers())
    refs[idx] = @spawnat pid foo(args...)
end
responses = [fetch(r) for r in refs]
```

String interpolation When doing I/O, forming intermediate strings is a source of overhead. Instead of

```
println(file, "$a $b")
```

do a direct output string by string using

```
println(file, a, " ", b)
```

Preallocate data When working with arrays or some other complex type, preallocate the space for data before it is actually needed. Allocations and garbage collections are substantial bottlenecks. Instead of

```
function xinc(x)
    return [x, x+1, x+2]
end
```

use

```
function xinc!{T}(ret::AbstractVector{T}, x::T)
    ret[1] = x
    ret[2] = x+1
    ret[3] = x+2
    nothing
end
```

Memory access ordering Julia stores arrays in column-major order. Not respecting this leads to cache misses and poor performance.
Avoid global variables Their value and type can be changed at any time, and this makes it harder for the compiler to optimize the code.
Use @time and @allocated The `@time` macro can measure time spent executing a function and also reports memory allocated by the function. Unexpected allocations often indicate a problem with type-stability. Similarly, the `@allocated` macro also reports allocated memory by an expression.
Profiler Use Julia's profiler to identify bottlenecks in your code.
Type declarations Julia is not the type of language where optional type declarations are the principal way to make code run faster. However there are a few exceptions. With user defined type like

```
type Foo
    field
end
```

the compiler will generally not know the type of `Foo.field`, since the value it refers to might be modified at any time. It helps to declare it as `field::Float64`. Similarly, it may help the compiler if the type in an expression is specified explicitly:

```
x = a[1]::Int32
```

Type stability Performance can significantly degrade when frequently used variables change their type. For example if the variable is supposed to be a float, declare it as `x = 1.0` or `x::Float64 = 1`, not `x = 1`.

6

External Programming Language Integration

In this chapter, Julia's ability to integrate, interpret and run external commands and libraries is presented.

6.1 External commands

Julia uses the same backtick command notation as shell, Perl and Ruby. However, writing one of the commands below

```
‘echo hello‘
‘perl -le ’$|=1; for (0..3) { print }’‘
pipeline(‘cut -d: -f3 /etc/passwd‘, ‘sort -n‘, ‘tail -n5‘)
```

differs in several aspects. The command is **only** created as an object, not run. When run explicitly (using `run(command)`), the output is not implicitly captured by Julia, but is printed to `stdout`. The command is never run in shell, Julia parses the command directly and runs it as its child process.

6.2 Calling C code

C functions can be called directly from Julia (even from the interactive prompt) without any generation or compilation, just by doing a call using `ccall`. The code has to be available as a shared library (compiled with `-shared -fPIC in GCC)`.

Libraries and functions are referenced using a tuple of the form

```
(:function, "library")
```

To actually generate a function call, `ccal() is used`:

```
t = ccall( (:clock, "libc"), Int32, ())
path = ccall((:getenv, "libc"), Ptr{UInt8}, (Ptr{UInt8},), "SHELL")
```

It is also possible to call any Julia's function in C code using C function pointers and `cfunction`, which accepts three arguments, the name of the Julia function, return type and a tuple of arguments:

```
function mycompare{T}(a::T, b::T)
    return convert(Cint, a < b ? -1 : a > b ? +1 : 0)::Cint
end
const mycompare_c = cfunction(mycompare, Cint, (Ref{Cdouble}, Ref{Cdouble}))
```

Mapping argument types between Julia and C has to be specified explicitly and is critical for correct functionality of the code.

Memory allocation and deallocation has to be handled by calls to the appropriate cleanup routines of the library used.

Global variables exported by native libraries can be accessed using the `cglobal` function:

```
cglobal((:errno,:libc), Int32)
```

Julia is not thread safe to multi-threaded C applications. As a precaution, a two-layered system has to be set up by passing a function to `SingleAsyncWork`.

```
cb = Base.SingleAsyncWork(data -> my_real_callback(args))
```

Limited support for C++ is provided by the `Cpp`, `Clang`, and `Cxx` packages.

7 Parallel Programming

Most modern computers possess more than one CPU, and several computers can be combined together in a cluster. In order to reach competitive level of overall performance, the support and implementation of task-level parallelism is necessary. The C/C++ language offers standards such as OpenMP and MPI, which allow high levels of parallelism and are the gold standard for high performance computing on clusters and supercomputers.

Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once. Julia's implementation of message passing is different from other environments such as MPI. The way Julia communicates is generally "one-sided", meaning that programmers have to explicitly manage only one side of a two-side communication, and these operations do not look like "send" or "receive", but rather like a higher-level function calls.

Julia's parallel programming is built on two main primitives: *remote references* and *remote calls*. A remote reference is an object, that can refer to any other object stored on a particular process, and can be used from any process. A remote call is a request by one process to call a certain function with certain arguments on another (and possibly the same) process, and a remote reference to the result is returned. Remote calls are non-blocking, meaning that the caller immediately proceeds to its next operation and the remote call happens somewhere else. It is possible to wait for the remote call to finish by calling `wait()` on its remote reference, and the full result can be obtained by calling `fetch()`. The idea is that parallel programming in Julia is asynchronous, meaning that instead of statements happening one after each other in a given order, they can be executed and/or finished in a different order and the programmer is not sure when.

The syntax is:

```
r = remotecall(processIndex, function, functionArg1, functionArg2...)
```

where `r` is a remote reference storing the result.

Each process has an associated identifier. The process providing the interactive Julia prompt always has an `ID` equal to 1. The processes used by default for parallel operations are referred to as *"workers"*. When there is only one process, process 1 is considered a worker. Otherwise, workers are considered to be all processes other than process 1.

The base installation of Julia supports two type of clusters:

- A local multicore machine with shared memory (similarly to OpenMP in C/C++)

- A cluster spanning machines with distributed memory (similarly to MPI in C/C++)

For adding, removing and assigning, functions `addprocs()`, `rmprocs()` and `workers()` are available.

7.1 Data Movement

Communication overhead very often is the most time consuming part of the whole parallel execution. Reducing the number of messages, the amount of data or optimizing the communication pattern is then critical to achieve good performance and scalability.

Basic data movement operations are `fetch()`, which moves an object to the local machine, and `@spawn`, which sends data or tasks to another process. In the example below, a random matrix is constructed and squared on another process and then moved to the local machine:

```
Bref = @spawn rand(1000,1000)^2
fetch(Bref)
```

On the other hand, a parallel flip-coin-counting Monte Carlo simulation, where data movement is not required, may look like this:

```
function count_heads(n)
    c::Int = 0
    for i=1:n
        c += rand(Bool)
    end
    c
end

a = @spawn count_heads(100000000)
b = @spawn count_heads(100000000)
fetch(a)+fetch(b)
```

In this programming pattern, many iterations are run independently on two processes and the result is combined together locally (e.q. operation *reduction)*. If we want to employ all the available processes, the parallel loop can be used:

```
nheads = @parallel (+) for i=1:200000000
  Int(rand(Bool))
end
```

7.2 Synchronization

The keyword `@sync` specifies a point in the code where each process must wait until all processes arrive to this point (similarly to `#pragma omp barrier` in OpenMP). In the example below, all processes synchronize before running another iteration of the loop.

```
@sync begin
    @parallel for i,j = 1:size(q,2)
        q[i,j] = q[i,j/2] + u[i,j]
    end
end
```

7.3 Channels

Channels are a fast way of inter-task communication. A `Channel(T::Type, n::Int)` is a shared queue of maximum length `n` holding objects of type `T`. Multiple readers can read off the channel via `fetch` and `take!`. Multiple writers can add to the channel via `put!`.

7.4 Shared Arrays

Shared Arrays use the shared memory to map the same array across many processes. Each process has access to every element of the array (unlike with `DistributedArrays`), where each process has its chunk of data. The constructor is of the form:

```
SharedArray(T::Type, dims::NTuple; init=false, pids=Int[])
```

This constructor creates a shared array of a bitstype `T` and size `dims`. Most algorithms work the same way on Shared Arrays as they do on normal arrays.

7.5 Cluster Managers

The networking management of Julia processes in a cluster environment is done via cluster managers. These managers take care of:

- launching workers
- managing workers' events
- optionally data transport

Only the master process with pid can launch and remove workers. All processes can communicate with each other using built-in TCP/IP transport. After launching, each worker starts to listening on a free port. The cluster manager collect worker's information and makes it available to the master process. Master process then sets up TCP/IP connections and every worker is notified of other workers. A mesh network is established in a way that each worker connects to a worker with lower id.

Julia has two cluster managers, the `LocalManager` launches workers on the same host, and the `SSHManager` connects these hostnames together. The launching method has parameters:

```
function launch(manager::LocalManager, params::Dict, launched::Array,
    c::Condition)
    ...
end
```

`LocalManager` specifies the type of manager, `Dict` are all the keyword arguments passed to the process, `Array` appends one or more configuration objects and `Condition` variable to be notified as and when workers are launched.

7.6 Network topology

There are there main types of network topology (following arguments are passed as a keyword argument `topology` to `addprocs()`):

- `:all_to_all` - all workers are connected to each other
- `:master_slave` - only the master process has connection to the workers
- `:custom` - the launch method of the cluster manager specifies the topology, e.q. which workers are directly connected to others

8 Conclusion

This survey presented Julia as one of the modern dynamic languages focusing primarily on performance and compact source code implementation. The main emphasis was to present the performance of Julia compared to other languages on a set of microbenchmarks, basic syntax overview, parallel environment, which is essential for scaling to high levels of performance, performance tips, embedding libraries written in C language and code profiling.

Bibliography

1. Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *CoRR*, abs/1411.1607, 2014.
2. Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.
3. Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.