LUND UNIVERSITY

Parallel computing in Julia

Case study from Dept. Automatic Control, Lund University

Bagge Carlson, Fredrik

2019

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Bagge Carlson, F. (2019). *Parallel computing in Julia: Case study from Dept. Automatic Control, Lund University*. (Technical reports TFRT-7657). Department of Automatic Control, Faculty of Engineering LTH, Lund University.

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Parallel computing in Julia:

Case study from Dept. Automatic Control, Lund University

Fredrik Bagge Carlson

fredrikb@control.lth.se

LUND UNIVERSITY

Department of Automatic Control

Technical Report TFRT-7657
ISSN 0280–5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2018 by Fredrik Bagge Carlson
fredrikb@control.lth.se. All rights reserved.
Printed in Sweden.
Lund 2018

Abstract

This document outlines how to setup and run computations in parallel using Julia on a collection of remote computers, such as computers in a university lab. After the environment has been setup, only minor modifications to serially executed code is necessary to enable parallel execution.

Written for Julia version 1.0

1. Introduction

Julia [Bezanson et al., 2017] is a modern programming language designed with high-performance numerical computing in mind. As such, it has stellar support for distributed computing. This document will focus on distributed computing using *workers* (multi-core), as opposed to shared-memory parallelism using threads (multi-thread). A worker is a separate instance of Julia, running either on the same machine or on a remote machine. This style of distributed computing has both benefits and drawbacks compared to multi-threading. The benefits include automatic thread safety and the obvious benefit of making use of the processing power of multiple different machines. The drawbacks include communication and memory overhead, and a task where several light computation loads are to be executed in parallel can oftentimes see the greatest speedup from multi-threading.

To set the stage, we briefly describe how to think about a worker. Fist of all, Julia's distributed computing functionality lives in the standard library `Distributed`. To start a additional workers, one can either start Julia with the command-line flag `-p`, or call the function `addprocs` at runtime. Workers can be started either on the local machine, to make use of all available processor cores, or on remote machines, such as lab computers or dedicated computing servers available via SSH.

The machine that starts additional workers is called the host. Computations can be assigned to any available worker by the host, provided that all required code is loaded at the assigned worker. A statement like `using Package` loads code on the host, but not on any workers. To run code on all workers, the macro `@everywhere using Package` is provided. Usage of this is demonstrated further in Sec. 3.1. Only workers started while `@everywhere` was called will load the code. Subsequently loaded workers are oblivious to this code. The same goes for variables and functions defined on the host, they must be defined `@everywhere` to be available on a worker. Common patterns for performing distributed computations are provided in Sec. 3.2.

2. Setup

In order to perform distributed computing on remote machines, the environment has to be setup on each machine. If you intend to run on your local machine only, you can skip this section. Below is an example of this procedure.

1. Verify that all computers you are interested in running on have the same Julia version installed. Julia will be launched from the same path as on the host computer (can be overridden with `exename` arg. to `addprocs`).

2. Ensure that you can perform password-less ssh to all computers (instructions).

3. In order to install all required packages on the remote machines, it is recommended to create a `Project.toml` file. Julia's package manager can create this file for you:

```
julia> cd("myproject") # Navigate Julia to the directory containing your code files. This
↪  can be done either before starting Julia or inside julia like this
julia> using Pkg
julia> pkg"activate ." # Set the current directory as the active Julia environment
julia> pkg"add LinearAlgebra Statistics DSP" # Add required packages, these are just
↪  examples
Resolving package versions...
Updating `/tmp/myproject/Project.toml`
[717857b8] + DSP v0.5.2
[37e2e46d] + LinearAlgebra
[10745b16] + Statistics
Updating `/tmp/myproject/Manifest.toml`
[621f4979] + AbstractFFTs v0.3.2
```

The commands above created the files `Project.toml` and `Manifest.toml`. They contain information about the packages required to run your code. At any time, the package environment you had when you created that file can be instantiated by the command `Pkg.instantiate()`. The difference between the manifest and project files are described in the documentation.

4. Initiate workers by running (the example starts 4 workers on each of the computers philon-02 to philon-12)

```
julia> addprocs([(@sprintf("philon-%2.2d",i),4) for i in 2:12], topology=:master_worker)
```

`master_worker` topology is recommended unless the workers have to communicate with each other.

5. The required packages are installed on remote machines by instantiating the project:

```
julia> using Distributed

julia> addprocs(["heron-01"]) # Start one worker on the remote machine heron-01
1-element Array{Int64,1}:
 2
julia> @everywhere using Pkg

julia> @everywhere pkg"activate ."  # Activate the current directory (myproject)

julia> @everywhere pkg"instantiate" # This installs all required packages
  Updating registry at `~/.julia/registries/General`
  Updating git-repo `git://github.com/JuliaRegistries/General.git`
    Fetching:  From worker 2: Updating registry at `~/.julia/registries/General`
From worker 2: Updating git-repo `https://github.com/JuliaRegistries/General.git`

julia> @everywhere pkg"precompile"
Precompiling project...
From worker 2:        Precompiling project...
```

This only works if every worker can find `myproject`, i.e., the path exists and is accessible on every machine. The default path of the workers can be specified with the `dir` arg. to `addprocs`, the default is the current path of the host.

3. Distributed computing

3.1 Loading code on remote machines

Julia is now ready to run your computations in parallel. Only code loaded on a worker can be run by that worker. Code is loaded on a worker by the macro `@everywhere`, e.g.:

```
@everywhere a = 2 # a is now = 2 on all loaded workers

@everywhere include("setup_computations.jl") # The files is included on all workers, note that
↪  the file must be available on every computer

@everywhere function myfun(a)
   a + 1
end # The function myfun is defined on all workers

@everywhere begin
   some_function_call()
   some_variable = something
end # All code in the block is run on all workers
```

If you start new workers after having run something `@everywhere`, you need to rerun that code on the new workers. *Note:* Before you run a `using` statement on remote workers, you have to run it on the host once for precompilation to take place, otherwise you will get an error (`WARNING: can only precompile from node 1`), hence the call to `precompile` in the example above.

If you need to include a file that is not available at the remote machine, such as a file located in your home directory not being available from the cloud computers, use the following include function

```
function include_remote(path, workers=workers(); mod=Main)
   open(path) do f
      text, s = read(f, String), 1
      while s <= length(text)
         ex, s = Meta.parse(text, s) # Parse text starting at pos s, return new s
         for w in workers
            @spawnat w Core.eval(mod, ex) # Evaluate the expression on workers
         end
      end
   end
end
```

This function reads the code into the variable `text` and performs an `eval` on the remote workers. An optional module can be specified, with `Main` as default.

3.2 Performing calculations on remote machines

One particular pattern that is suitable for parallel processing is Monte-Carlo simulations and calculations. To launch, e.g., many Monte-Carlo computations in parallel, a pattern like this is useful:

```
@everywhere include("setup_computations.jl")
all_results = pmap(1:number_of_montecarlo_runs) do index
   result = perform_computation(index)
end
```

`pmap` is a parallel map operation that automatically selects workers to perform the computations on. The index variable will take the numbers `1:number_of_montecarlo_runs` and can be used to, e.g., set the random seed or something similar. The function `perform_computation(index)` was defined in the script `setup_computations.jl` that was loaded in the beginning. The variable `all_results` will be a vector of length `number_of_montecarlo_runs` containing the results of the individual runs of the map body.

If the computations are not suitable to launch from a loop, one can launch computations on a remote worker with

```
f1 = @spawn run_some_computation() # Run computation on automatically chosen worker
f2 = @spawnat 3 run_some_other_computation() # Run computation on worker 3
```

`f1` and `f2` are of type `Future`, and the results must be fetched before used

```
result1 = fetch(f1) # This call blocks until computation of f1 is done
result2 = fetch(f2)
```

Another useful pattern for launching computations, if one is not comfortable with the map operation, is the following:

```
futures = Vector{Future}(num_iterations) # Create vector to hold all Futures
for iteration = 1:num_iterations
   f = @spawn perform_computation(iteration)
   futures[iteration] = f
end
results = fetch.(futures) # The dot . broadcasts the function call over the vector
```

For-loops can also be distributed with the macro `@distributed`, that accepts an optional reduction function, e.g.:

```
julia> @distributed vcat for i = 1:5
           myid() # Returns the id of the worker
       end
5-element Array{Int64,1}:
 2
 2
 3
 4
 5
```

```
julia> @distributed hcat for i = 1:5
            myid()
       end
 1×5 Array{Int64,2}:
  2  2  3  4  5

julia> @distributed (+) for i = 1:5
           myid()
       end
16
```

Distributed for-loops are to be preferred when the calculation involves reduction of many small results (like summing up numbers), whereas parallel maps are to be preferred when a vector of large results is desired:

```
julia> @time a = @sync @distributed vcat for i = 1:10 # Creating a result vector
            zeros(1000,1000)
       end;
2.350288 seconds (198.40 k allocations: 161.418 MiB, 5.28% gc time)

julia> @time b = pmap(1:10) do i  # Creating a result vector (the preferred way)
            zeros(1000,1000)
       end;
0.882909 seconds (220.28 k allocations: 86.244 MiB, 9.48% gc time)
##########################################################################################
julia> @time sum(pmap(1:10000) do i # Reducing with +
            myid()
       end)
0.543199 seconds (856.46 k allocations: 34.684 MiB, 2.39% gc time)

julia> @time a = @sync @distributed (+) for i = 1:10000 # Reducing with + (the preferred way)
            myid()
       end;
0.054469 seconds (52.08 k allocations: 2.533 MiB)
```

4. Getting results back

If you launch Julia from a remote computer, but want to analyze the results of the parallel computations on, e.g., your office computer, then

1. Place your script file in a mounted location, e.g., `/work/$USER` (preferable since the file saved below might become large) or `/home/$USER`. For simplicity, navigate to this folder on both local and remote machine before starting Julia.

2. Run `open(file->serialize(file, results), "res.bin","w")` to save the results to a binary file called `res.bin`.

3. On your office computer, run `results = open(deserialize, "res.bin")` to load the results. If the office computer and the remote computers are running different Julia versions, loading of the file might not work, in that case, use a package like JLD.jl or BSON.jl (recommended) to save and load the results instead.

5. Miscellaneous

How to figure out which packages to install on remote computers All the packages that you are calling `using PackageName` on.

How many workers to launch The optimal is typically to utilize all *physical* cores on each machine. Some operations, like matrix operations etc., automatically run in parallel, in which case you will see limited speedup from launching more than a single worker per machine. If you are running in a lab full of students, it may be good to limit the number of workers to 1 or 2 per machine to not slow it down too much.

Order of computations If the computations you run have vastly different runtimes, try to launch the longest running computations first, e.g.:

```
pmap(10:-1:1) do i
        sleep(i)
end
```

will finish faster than

```
pmap(1:10) do i
   sleep(i)
end
```

Host machine workers You can launch workers on the host machine as well with the command `addprocs(4)`. This is useful if 1) You have no remote machines. 2) You want MORE POWER. Be sure to do this *after* adding the remote workers if you want to use both.

Startup script Note that workers do not run a `startup.jl` script, nor do they synchronize their global state (such as global variables, new method definitions, and loaded modules) with any of the other running processes.

Non-Julia dependencies These can be a bit tricky to handle. All dependencies have to be configured at every remote machine. If your computations are native Julia only, or installed automatically as part of `Pkg.add()`, you're safe. If not, I would ask the system administrator to help out.

Sending data between workers You may find the package ParallelDataTransfer.jl useful. It allows you to send variables between workers, in particular, from the host to the remote machines.

The result of a parallel computation Keep in mind that the result of, e.g., a `pmap` statement is automatically sent from the worker to the host. If this result is large, this communication can become a bottleneck, e.g.:

```
julia> sizeof(zeros(10_000,1_000)) ÷ 1e6
80.0 # Mb
```

6. Troubleshooting

WARNING: Node state is inconsistent: node failed to load cache from /var/tmp/username/lib/*.ji. If you get this message, it might be due to the host computer and the remote computer running different versions of LLVM.

WARNING: can only precompile from node 1 First time you call `using Package` must be on the host only, i.e., not inside an `@everywhere` statement.

7. Documentation

- Julia manual
- Julia parallel computing manual
- Standard Library (Distributed)

References

Bezanson, J., A. Edelman, S. Karpinski, and V. B. Shah (2017). "Julia: a fresh approach to numerical computing". *SIAM Review* **59**:1, pp. 65–98.