How to run Julia computations on the lab computers

Fredrik Bagge Carlson

January 4, 2018

Abstract

This document outlines how to setup and run computations in parallel using the lab computers and Julia. With this simple method, one can see close to a (number of labs $\times$ number of machines $\times$ number of cores) $= (3 \times 12 \times 4) = 144$ fold speedup of parallel computations.

1 Setup

- 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.
- Ensure that you can perform password-less ssh to all lab computers Instructions.
- Install all relevant julia packages at the same location on all computers. Installing on a remote disc (like /home or /work) is not advised. I created the following script (`install_packages.jl`) which was placed in my home folder

```julia
Pkg.init()
Pkg.add("Flux"); using Flux
Pkg.add("ValueHistories"); using ValueHistories
Pkg.add("IterTools"); using IterTools
Pkg.add("MLDataUtils"); using MLDataUtils
Pkg.add("DynamicMovementPrimitives"); using DynamicMovementPrimitives
Pkg.add("OrdinaryDiffEq"); using OrdinaryDiffEq
```

  After that you run the following in the terminal

```
allclients -o -g ktesibios -g philon -g heron "export JULIA_PKGDIR=/var/tmp/$USER;
↪  julia install_packages.jl"
```

- Start julia from a computer running the same version of julia (and LLVM) as the lab computers. In my case, my office computer tends to run a newer version, why I have to start julia on one of the lab computers via ssh.
- Initiate workers by running (the example starts 4 workers on each of the computers philon-02 to philon-12)

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

2 Performing computations

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,

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

@everywhere include("setup_computations.jl") # The files is included on all workers

@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
```

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'll get an error (`WARNING: can only precompile from node 1`), hence all the `using` statements in the install script.

To launch, e.g., many Monte-Carlo computations in parallel, I typically use a pattern like this

```julia
@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` loaded in the beginning. The variable `all_results` will be a list 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

```julia
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

```julia
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

```julia
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
```

3 Getting results back

If you launch julia from a lab computer but want to analyze the results of the parallel computations on your office computer, then

- 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.
- Run `open(file->serialize(file, results), "res","w")` to save the results to a binary file called `res`.
- On your office computer, run `results = open(file->deserialize(file), "res")` to load the results. If the office computer and the lab computers are running different julia versions, loading of the file might not work, in that case, use the package JLD to save and load the results instead.

4 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, which currently is 4 per computer. 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 the lab is full of students, I usually limit my 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 computations first, e.g.,

```julia
pamp(10:-1:1) do i
   sleep(i)
end
```

will finish faster than

```julia
pamp(1:10) do i
   sleep(i)
end
```

unless the number of workers is greater than 9.

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.

.juliarc.jl Note that workers do not run a .juliarc.jl startup 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.

Dependencies These can be a bit tricky. 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 Anders Blomdell to help out.

More computers You might be able to run computations on `cloud-{01-07}` as well.

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.

5 Documentation

- Julia manual
- Standard Library (parallel)