Parallel computing in Julia

Fredrik Bagge Carlson
`fredrikb@control.lth.se`

Outline

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

Introduction

Julia is a modern programming language designed with high-performance numerical computing in mind. As such, it has stellar support for distributed computing.

Introduction

- Multi-core – Multiple workers
- Multi-thread – Multiple threads on same worker, shared memory.

Multi-core vs. multi-thread

+ automatic thread safety
+ making use of the processing power of multiple different machines
- communication and memory overhead

Workers

How to think about a worker?

- A worker is a separate instance of Julia.
- Each worker has it's own memory.
- Each worker only knows things defined explicitly in its julia session.

Workers

- 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.
  - Local machine
  - Remote machines (e.g., lab computers, cloud)
- The machine that starts workers is the *host*.

Code availability

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.
- `@everywhere using Package` loads package on all workers.
- Only workers started while `@everywhere` was called will load the code.

Environment setup

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.

1. Verify that all computers have the same Julia version installed. Julia will be launched from the same path as on the host computer.
2. Ensure password-less ssh.
3. To install all required packages on the remote machines, it's recommended to create a `Project.toml` file:

```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
using Pkg
pkg"activate ." # Set the current directory as the active Julia
↪  environment
pkg"add LinearAlgebra Statistics DSP" # Add required packages, these
↪  are just examples
```

Environment setup

1. Initiate workers by running (example)

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

   `master_worker` is recommended unless communication required.

2. The required packages are installed remotely by instantiating the project:

   ```
   using Distributed
   addprocs(["heron-01"]) # Start one worker on heron-01
           # 1-element Array{Int64,1}:
           # 2
   @everywhere using Pkg
   @everywhere pkg"activate ."  # Activate current dir (myproject)
   @everywhere pkg"instantiate" # This installs required packages
   Updating registry at `~/.julia/registries/General`
   Updating `git://github.com/JuliaRegistries/General.git`
   Fetching:  From worker 2: Updating registry
   From worker 2: Updating
    ↪  `https://github.com/JuliaRegistries/General.git`
   ```

Environment setup

```
@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.

Loading code on remote machines

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, 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.

Loading code on remote machines

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

```julia
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.

Performing calculations on remote machines

One particular pattern that is suitable for parallel processing is Monte-Carlo simulations and calculations. A pattern like this is useful:

```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.

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.

Performing calculations on remote machines

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)
```

Performing calculations on remote machines

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

```
futures = Vector{Future}(num_iterations) # vector to hold Futures
for iteration = 1:num_iterations
    f = @spawn perform_computation(iteration)
    futures[iteration] = f
end
results = fetch.(futures)
```

Performing calculations on remote machines

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
```

Performing calculations on remote machines

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:

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

# Creating a result vector (the preferred way)
@time b = pmap(1:10) do i
    zeros(1000,1000)
end;
0.882909 seconds (220.28 k allocations: 86.244 MiB, 9.48% gc time)
```

Performing calculations on remote machines

```
@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)

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

Error handling

Any error stops pmap from processing the remainder of the collection. To override this behavior you can specify an error handling function via argument `on_error`

```
julia> pmap(x->iseven(x) ? error("foo") : x, 1:4; on_error=ex->0)
4-element Array{Int64,1}:
 1
 0
 3
 0
```

Errors can also be handled by retrying failed computations. Keyword arguments `retry_delays` and `retry_check`.

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/$USER` or `/home/$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.
4. 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.

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. If you are running in a lab full of students...

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
```

Miscellaneous

Host machine workers You can launch workers on the host machine as well with the command `addprocs(4)`. Be sure to do this *after* adding the remote workers if you want to use both.

Startup script 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. I would ask the system administrator to help out.

Sending data between workers ParallelDataTransfer.jl is useful. It allows you to send variables between workers.

The result of a parallel computation 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
```

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.

Documentation

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

Multi threading

`Base.Threads`

+ Threads have significantly lower overhead compared to workers.
- You need to worry about thread safety (you really do).

Not thread safe by default

- IO (e.g., `println`)
- Random
- Regex

How to thread

Set environment variable `JULIA_NUM_THREADS`, e.g., by placing `export JULIA_NUM_THREADS=4` in `.bashrc` (Atom defaults to number of cores in your machine).

```
Threads.@threads for i = 1:N
    do_work(i)
end
```

- Loop iterations must be independent

Threaded random number generation

Each thread must have it’s own random number generator.

```
r = [MersenneTwister(1) for _ in 1:Threads.nthreads()]
function foo(r)
   @threads for i in 1:1000
      rng = r[threadid()]
      a = randn(rng, N)
   end
end
```

The same goes for regex.

Mutable arguments

Be wary of calling functions that modify their input arguments! Use separate workspaces for each thread.

```julia
struct Workspace{T}
    xh::Vector{T}
    x::Vector{T}
    y::Vector{T}
    w::Vector{T}
    ⋮
end

workspaces = ntuple(i->Workspace(), 4) # Works if you provide argument-free
↪  constructor
RAN = ntuple(MersenneTwister, 4))
Threads.@threads for i = 1:mc
        ws  = workspaces[threadid()]
        rng = RAN[threadid()]
        xh  = pf!(rng, ws, N, g, f, σw0)
end
```

Reading

- Parallel computing in Julia : Case study from Dept. Automatic Control, Lund University 2019
- Julia parallel computing manual
- Standard Library (Distributed)