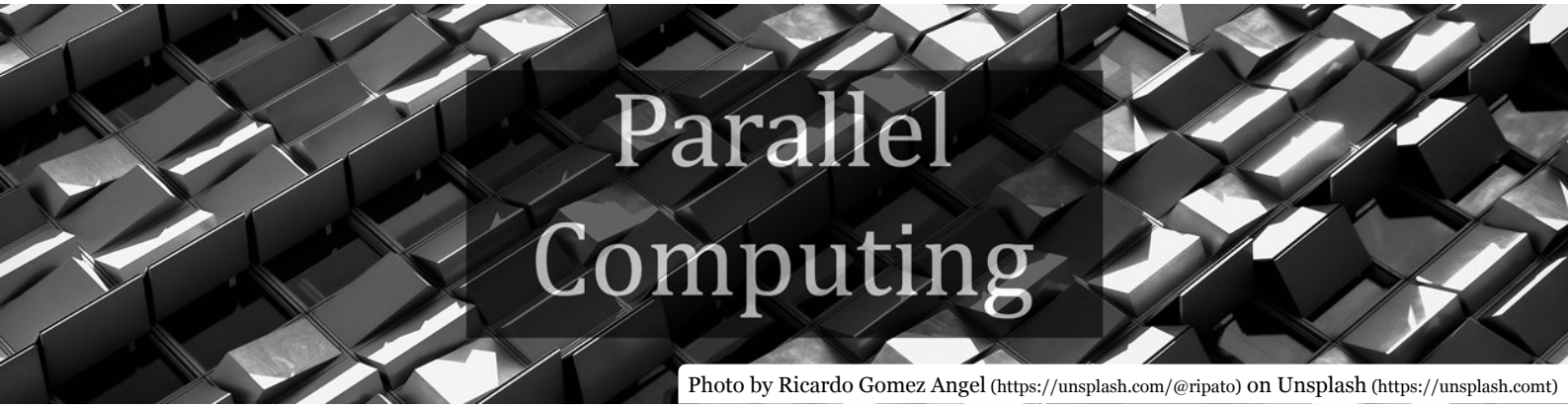

Photo by Ricardo Gomez Angel (https://unsplash.com/@ripato) on Unsplash (https://unsplash.comt)

Parallel computing

11 minute read

In this lesson we will deal with **parallel computing**, which is a type of computation in which many calculations or the execution of processes are carried out simultaneously on different CPU cores. We will show the differences between **multi-threading** and **multi-processing** and we will learn how those techniques are implemented in Julia.

For this lesson you will need Julia version 1.3 or above.

When to use parallel computing

If you need to perform many operations independent from each other (i.e. mapping a function over an array), parallel computing will give a huge improvement in speed to the computation.

Monte Carlo simulations (https://en.wikipedia.org/wiki/Monte_Carlo_method) and matrix operations are other kind of computations which benefit greatly from parallel computing.

Multi-threading and Multi-processing

Both techniques can be used to achieve parallel processing, but they have their pros and cons.

- **Threads are lightweight**, since all the threads share the same space in memory, and you can launch as many threads as you want. The CPU will run simultaneously a number of threads equal to the number of cores. Since the memory is shared you must **pay attention** to correct **synchronisation** when multiple threads are accessing the same variable. Julia introduces **locks** (https://docs.julialang.org/en/v1/base/multi-threading/) to prevent this kind of racing condition. A downside of multi-threading is that **all the threads must live in the same physical machine**.
- **Multi-processing** is the right choice if we want to run computations on a cluster (a series of different physical machines). **A process has its own memory space** and thus will not have synchronisation issues, the downside is that starting a process may be slow and **each process consumes more memory than a thread**. Although multi-processing requires more memory than multi-threading, it is possible to share memory between process and if we work with a cluster, it is possible to manage arrays and matrices much bigger than what a single machine would be able to do.

In general, as a rule of thumb, we will use **threads** if we plan to run our code on a **single machine** and **processes** if the code must be able to run on a **cluster** of machines.

Multi-threading

In order to use multi-threading we need to start Julia with a number of threads equal to the number of you CPU cores. If you are using the Juno IDE it will automatically start Julia with the appropriate number of threads. If you are working from the REPL, you need to manually start Julia from the command line. My laptop has 4 cores and I use Windows, so I need to start Julia with 4 threads. To start Julia from the command line, navigate to the folder which contains the Julia bin and type:

```
set JULIA_NUM_THREADS=4
julia.exe
```

If you are using Unix, type the following code instead:

```
export JULIA_NUM_THREADS=4
julia
```

> For this guide I recommend using the Juno IDE as it makes starting threads much easier.

You can check the number of available threads with the following Julia code:

```
>>>Threads.nthreads()
4
```

Let's suppose we want to compute the map of the Bessel function `besselj1` (https://juliamath.github.io/SpecialFunctions.jl/stable/functions_list/#SpecialFunctions.besselj1) over an array and store the results, we can do it regularly in the following way using broadcasting:

```
using Pkg
Pkg.add("SpecialFunctions")

using SpecialFunctions

x = range(0,100, length=10000)

results = zeros(length(x))

results .= besselj1.(x)
```

It is possible to measure the execution time of a function using the macro `@btime` from the `BenchmarkTools` package. To use it we need to first install `BenchmarkTools`.

```
using Pkg
Pkg.add("BenchmarkTools")
using BenchmarkTools
```

And to measure the execution time of `besselj1` we need to run the following code:

```
>>>@btime results .= besselj1.(x)
674.800 μs (3 allocations: 192 bytes)
```

The computation takes 674.800 μs on my machine. Although it is not the fastest solution, it is possible to rewrite this operation using a for loop:

```
for i in 1:length(x)
    results[i] = besselj1(x[i])
end
```

This operation takes 1.835 ms on my machine to complete. The difference in time is due to the fact that the second version has to read the values of `x` from the array at each iteration. If we were to use a slower function the two implementations would be equivalent:

```
function slow_func(x)
    sleep(0.005)
    return x
end

y = 1:0.1:10
res = zeros(length(y))

>>>@btime res .= slow_func.(y)
593.243 ms (643 allocations: 15.39 KiB)

>>>@btime for i in 1:length(y)
        res[i] = slow_func(y[i])
    end
590.302 ms (823 allocations: 19.50 KiB)
```

Coming back to the example with `besselj1`, in that loop every iteration is independent from the next one: this hints the possibility to make the code parallel. To achieve parallelization, we import the `Threads` module and call the `@threads` macro:

```
using Threads
@threads for i in 1:length(x)
    results[i] = besselj1(x[i])
end
```

Timing this block of code, I find that the execution time is 396.900 µs. As you can see it is roughly 4 times faster than the equivalent code without multi-threading and is even faster than the first option.

Composable multi-threading

Starting from Julia 1.3, it is possible to start an operation on a thread with the `Threads.@spawn` macro. At the time of writing this feature is considered experimental and as such will probably see some changes in the interface. With the `@spawn` macro it is possible to start several tasks simultaneously and then fetch the results. For example it is possible to implement the classic highly-inefficient tree recursive implementation of the Fibonacci sequence using multi-threading

```
1  import Base.Threads.@spawn
2  using BenchmarkTools
3
4  function fib(n::Int)
5      if n < 2
6          return n
7      end
8      t = fib(n - 2)
9      return fib(n - 1) + t
10 end
11
12 function fib_threads(n::Int)
13     if n < 2
14         return n
15     end
16     t = @spawn fib_threads(n - 2)
17     return fib(n - 1) + fetch(t)
18 end
```

At line 4 we see the classical Fibonacci recursive function and at line 12 a recursive Fibonacci function. The only difference between the two implementation is the `@spawn` at line 16 and the `fetch` at line 17.

Although in this case there is no gain in performance (the multi-threaded version is slower) this is an example of how it is possible to write complex and nested functions which spawn tasks on threads.

A simpler example may be this one: let's consider the case where we want to compose the result of some slow function calls, we can spawn each function call on a different thread and then compose the result:

```
1  import Base.Threads.@spawn
2  using BenchmarkTools
3
4  function slow_func(x)
5      sleep(0.005) #sleep for 5ms
6      return x
7  end
8
9  @btime let
10     a = @spawn slow_func(2)
11     b = @spawn slow_func(4)
12     c = @spawn slow_func(42)
13     d = @spawn slow_func(12)
14     res = fetch(a) .+ fetch(b) .* fetch(c) ./ fetch(d)
15 end
16
17 @btime let
18     a = slow_func(2)
19     b = slow_func(4)
20     c = slow_func(42)
21     d = slow_func(12)
22     res = a .+ b .* c ./ d
23 end
```

At line 10-13 we spawn `slow_func` at different threads and at line 14 we combine the results. On the contrary, from line 18 to 22 we perform the exact same computation without using `@spawn`. The result of the timings is respectively 5.484 ms and 24.231 ms, the multi-threaded function is much faster!

On the other hand, if we use multi-threading for faster functions (such as the sine function) the results are not as good:

```
1  @btime let
2      x = 1:100
3      a = @spawn sin(2)
4      b = @spawn sin(4)
5      c = @spawn sin(42)
6      d = @spawn sin(12)
7      res = fetch(a) .+ fetch(b) .* fetch(c) ./ fetch(d)
8  end
9
10 @btime let
11     x = 1:100
12     a = sin(2)
13     b = sin(4)
14     c = sin(42)
15     d = sin(12)
16     res = a .+ b .* c ./ d
17 end
```

In the first case (line 1-8) I get 3.525 µs of execution and on the second case (line 10-17) I get 1.499 ns. For this example, it is not convenient to use multi-threading as the overhead due to spawning new threads and fetching the results is greater than the performance gain due to parallelization.

For more information on composable multi-threading, see this (https://julialang.org/blog/2019/07/multithreading) article.

> Always test if using multi-threading and in general parallel computation will yield performance gains in your case.

Multi-processing

In order to use multiprocessing, we need to first install and include the `Distributed` package. Before running the following code please **restart the REPL**.

```
1 using Pkg
2 Pkg.add("Distributed")
3
4 using Distributed
```

Now we need to add a number of process equal to or smaller than the number of CPU cores. In my case, my CPU has four cores, thus I will add four processes using `addprocs`:

```
1 addprocs(4)
```

There are two macros available in Julia to spawn a process: `@spawn` and `@spawnat`. `@spawn` will run the desired command in the first available process, while `@spawnat` will run the instruction on the desired process. Let's see how they work with an example:

```
1 >>>fetch(@spawn myid())
2 2
3 >>>fetch(@spawnat 3 myid())
4 3
```

At line 1, `fetch` is used to retrieve the result of the remote computation. The result printed at line 2 will vary between subsequent runs, as the function `myid` (which returns the id of the process) will run every time on a different process. On the contrary, when we call `@spawnat 3 myid()` we will always get `3` as the result, since we have chosen to run `myid()` on the third process.

Loading code on a process

In the case of multi-processing, since each process has its own memory address and will run on a different CPU core, if we want to be able to run a function on a certain process we need to first define that function on that process.

There is a convenient macro in Julia used to define “something” on all the process: the `@everywhere` macro. Once a function is defined on a process, it is possible to spawn a computation task on that process.

Let’s suppose we want to compute the `gamma` function which is available through the `SpecialFunctions` package. First we need to import the package `@everywhere`, then it becomes possible to call the `gamma` function on other processes:

```
@everywhere using SpecialFunction

res = fetch(@spawn gamma(5))
```

The same procedure applies to user defined functions: if a function has to be computed on a process, it has to be defined `@everywhere`:

```
@everywhere function my_func(x)
    return x^3*cos(x)
end

fetch(@spawnat 2 my_func(0.42))
```

Parallel mapping

Multi-processing is extremely useful when it comes to mapping a function over an array.

There are two options to perform a mapping: you can either use a `@distributed` for loops or use the `pmap` function.

In order to use distributed for loops, we need to have a shared resource (a shared array) where the results of the computation can be stored. For this purpose, we use the package `SharedArrays`. A shared array is an array which is accessible and shared by all the process on a single machine.

```
using Pkg
Pkg.add("SharedArrays")

@everywhere using SharedArrays

res = SharedArray(zeros(10))

@distributed for x in 1:10
    res[x] = my_func(x)
end

>>>res
10-element SharedArray{Float64,1}:
    0.5403023058681398
   -3.3291746923771393
  -26.729797408212026
  -41.833191735271164
   35.45777318290328
  207.39678191647906
  258.58847323975345
  -74.49601731001013
 -664.2139609139294
 -839.0715290764524
```

Distributed for loops have the advantage of being lightweight, but they don't perform load balancing. `@distributed` divides into equal portions the for loop and assign each portion to a process, if the function which is being computed has different execution times depending on the argument this strategy may not be the best. In this case a better solution is to use `pmap` which implements a load-balancer.

pmap

`pmap` (https://docs.julialang.org/en/v1/stdlib/Distributed/index.html#Distributed.pmap) is a function which takes as its arguments the function which should be mapped (which has to be available to each processor) and the array/range on which it has to be mapped. For example:

```
@everywhere function my_func(x)
    return x^3*cos(x)
end

>>>pmap(my_func, 1:10)
10-element Array{Float64,1}:
    0.5403023058681398
   -3.3291746923771393
  -26.729797408212026
  -41.833191735271164
   35.45777318290328
  207.39678191647906
  258.58847323975345
  -74.49601731001013
 -664.2139609139294
 -839.0715290764524
```

The advantage of `pmap` is that it doesn't rely on `SharedArrays` and can be used also in the case of a computer cluster. By default, `pmap` will ask each process to compute one value at a time, when a process has finished computing the assigned value it will receive a new value to compute. In this way load balancing will be performed between all the processes and the computation will generally be faster.

In the case where the time require to transfer data between processes is much longer than the computation time, it may be a good idea to set the `batch_size` to a value greater than 1. In this one, each process will compute a batch of values before transferring the data. As a rule of thumb, it is a good idea to set the **batch size** so that the **computation time** of a batch is **at least 10 ms**, this way most of the time will be spent on the computation of values and not on data transfer. In general, if the total computation will take less than 100 ms, there's no need to use multi-processing.

```
using Distributed
using BenchmarkTools
addprocs(4)

@everywhere using SpecialFunctions
@everywhere function my_func(x)
    arg = repeat([x], 1000)
    return besselj1.(arg)
end

# %%

@btime my_func.(1:1000) # 91.332 ms

@btime my_func.(1:100) # approximatively 10 ms

@btime pmap(my_func, 1:1000, batch_size=1) # 88.203 ms
@btime pmap(my_func, 1:1000, batch_size=100) # 37.279 ms
```

Conclusions

In this lesson we have learned how to use multi-threading and multi-processing to achieve parallel computing. To summarise:

- Use **multi-threading** if you perform your computations on a **single machine**. **Threads are lightweight** and you can easily spawn tasks and parallel loops. Multi-threading is recommended if you have computations which take more than 100 µs
- Use **multi-processing** if you are working with a **cluster** or if your computations take longer than 100 ms. Remember that **spawning a process is an expensive operation**.

You can find more information about multi-processing on this guide (https://techytok.com/multiprocessing-in-julia-module/) and at the official documentation page (https://docs.julialang.org/en/v1/manual/parallel-computing/) about Parallel Computing.

If you liked this lesson and you would like to receive further updates on what is being published on this website, I encourage you to subscribe to the **newsletter** (https://techytok.com/newsletter/)! If you have any **question** or **suggestion**, please post them in the **discussion below**!

Thank you for reading this lesson and see you soon on TechyTok!

Tags: Julia

Categories: Lessons Tutorial

Updated: December 27, 2019

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Asim Hassan Dar • 6 months ago

A bit of a newbie question: what kind of computing is it when it's not parallel but with more cores? For example, I use MATLAB and when pretty much doing anything I can see that I am using a multitude of cores during computations (non-parallel, i.e no par for loops in MATLAB lingo) but when I use Julia i just get 100% usage of a single core—which is my current speed bottleneck.

• Reply • Share ›

aureamerio Mod ➚ Asim Hassan Dar • 6 months ago

Hello, by definition parallel computing is the process of performing a computation over several cores. Parallel computing can be achieved in different ways, either using multi-threading or multi-processing. Currently Julia performs computations by on a single process and on a single thread (mostly), and that is possibly the reason why you get 100% usage of a single core. On the other hand, while I'm not familiar with Matlab, I've read on some articles that it sometimes automatically performs some sorts of parallel computations internally, which may be your case.

Just a question: how can you be sure that Matlab is performing its computations on different cores simultaneously? When you run a program there will be also several other system processes from other applications still running, thus you will see some CPU usage on each of your cores. Unless you see 100% CPU usage on each core you cannot be sure that Matlab is using each and every core for your computations, and on the contrary it is possible that in your scripts Matlab uses mainly one core and not at full capacity (although this is just a guess, since I'm not a Matlab user).

If you write a Julia script using multi-threading, and perform some computationally expensive operations like matrix multiplications of big matrices, you will see al your cores run at 100% capacity.

If you want, you can tell us something more about what you want to achieve using parallel

see more

• Reply • Share ›

joe • 6 months ago

You haven't defined "slow_func"

• Reply • Share ›

aureamerio Mod ➚ joe • 6 months ago

thank you for pointing it out, I have updated the guide!

1 • Reply • Share ›

/