11c. Task-Based Parallelism: @spawn

Martin Alfaro
PhD in Economics

INTRODUCTION

The previous section explained the basics of multithreading. In particular, we've shown that operations can be computed either sequentially (Julia's default) or concurrently. The latter approach enables multiple operations to be processed simultaneously, with operations running as soon as a thread becomes available. When Julia's session is initialized with more than one thread, this implies that computations can be executed in parallel.

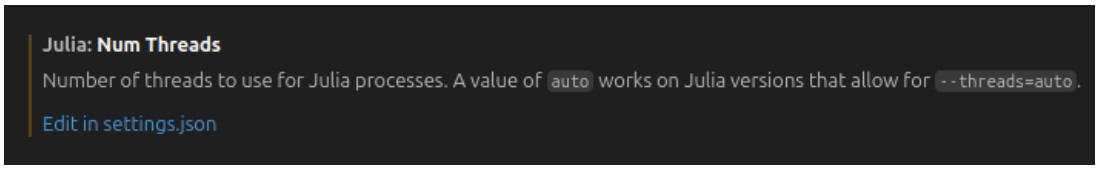
This section will focus on Julia's native multithreading mechanisms, a topic that will span several sections. Our primary goal here is to demonstrate how to write multithreaded code, rather than exploring how and when to apply the technique.

We've deliberately structured our explanation in this way to smooth subsequent discussions. However, a crucial caveat at this point remains necessary: while multithreading can offer significant performance advantages, it's not applicable in all scenarios. In particular, multithreading demands extreme caution in handling dependencies between operations, as mismanagement can lead to silent catastrophic bugs. After grasping a basic understanding of parallelism techniques, we'll touch on unsafe-thread operations.

ENABLING MULTITHREADING

Julia initializes every session with a given pool of threads available. Each of these threads is responsible for executing a given set of instructions. Consequently, the number of threads delimits the number of instructions that the CPU can handle simultaneously.

By default, Julia only operates with a single thread, requiring setting an alternative number of threads to enable multithreading. You can achieve this in VSCode or VSCodium by going to *File* > *Preferences* > *Settings*. Then, search for the keyword *threads*, prompting the following line:



After pressing *Edit in settings.json*, you should add the line `"julia.NumThreads": "auto"`. This will automatically identify the number of threads based on your computer's features, which will either match your CPU's logical or physical threads.

Notice the effects will take place after starting a new session. To check whether the effects have taken place, use the command `Threads.nthreads()`. This displays the number of threads available in the session. Any number greater than one will indicate that multithreading is activated. Notice also that the changes are permanent, so that every new Julia session will start with the number of threads specified.

Once we have a session with more than one thread, there are several packages for performing multithreaded computations. The focus on this section will be on the built-in package `Threads`, which is automatically imported when you start Julia.

```
# package Threads automatically imported when you start Julia

Threads.nthreads()
```

```
2
```

```
using Base.Threads      # or `using .Threads`

nthreads()
```

```
2
```

> **Warning!** - Loaded Package
> All the scripts below assume that you've executed the line `using Base.Threads`. Furthermore, all the examples are based on a session with **two worker threads**.

TASK-BASED PARALLELISM: @SPAWN

The first approach we'll cover is implemented through the macro `@spawn`, which streamlines the application of the previous section's techniques. Specifically, by prepending any operation with `@spawn`, we create a (non-sticky) task that's scheduled right away for its execution. Recall that once a task is scheduled, it'll immediately start its computation if there's a thread available.

Unlike other approaches that we'll present, `@spawn` requires explicitly instructing Julia to wait for the task to complete. The way to do this depends on the nature of the output. For tasks that perform computation and additionally return an output, we have the function `fetch`. This waits for calculations of a task to finish and then returns its output. Since parallel computation requires spawning multiple tasks, the function argument of `fetch` should comprise all the tasks spawned and `fetch` be broadcasted.

In the following, we illustrate `fetch` with two spawned tasks that return vectors as their output.

```julia
x = rand(10); y = rand(10)

function foo(x)
    a = x .* -2
    b = x .*  2

    a,b
end
```

```julia
x = rand(10); y = rand(10)

function foo(x)
    task_a = @spawn x .* -2
    task_b = @spawn x .*  2

    a,b = fetch.((task_a, task_b))
end
```

It's important to distinguish between `task_a` and `a`: while `a` refers to the vector created (i.e., the task's output), `task_a` denotes the task creating the vector `a`. The distinction is essential since the function `fetch` only takes a task as its input.

Alternatively, for operations that don't return any output, we can use either the function `wait` or the macro `@sync`. The function `wait` is applied similarly to `fetch`. Instead, the macro `@sync` requires wrapping all operations to be synchronized, which is done by enclosing the operations with the keywords `begin` and `end`.

For the demonstration, let's consider a mutating function. Mutating functions are suitable as an example, since they only modify values of a collection, without returning any output.

```julia
x = rand(10); y = rand(10)

function foo!(x,y)
    @. x = -x
    @. y = -y
end
```

```julia
x = rand(10); y = rand(10)

function foo!(x,y)
    task_a = @spawn (@. x = -x)
    task_b = @spawn (@. y = -y)

    wait.((task_a, task_b))
end
```

```julia
x = rand(10); y = rand(10)

function foo!(x,y)
    @sync begin
        @spawn (@. x = -x)
        @spawn (@. y = -y)
    end
end
```

MULTITHREADING OVERHEAD

To see the advantages of `@spawn` in action, let's compute the sum and maximum of a vector `x`, for which we present a sequential and a simultaneous approach. To clearly shed light on the benefits of parallelization, we also include the time to execute each operation in isolation. The results establish that the time of the sequential procedure is equivalent to the sum of each computation. Instead, thanks to parallelism, the execution time under multithreading is roughly equivalent to the maximum time required for either computation.

```julia
x = rand(10_000_000)

function non_threaded(x)
    a           = maximum(x)
    b           = sum(x)

    all_outputs = (a,b)
end
```

```
julia> @btime maximum($x)
  7.705 ms (0 allocations: 0 bytes)
julia> @btime sum($x)
  3.131 ms (0 allocations: 0 bytes)
julia> @btime non_threaded($x)
  10.917 ms (0 allocations: 0 bytes)
```

```julia
x = rand(10_000_000)

function multithreaded(x)
    task_a      = @spawn maximum(x)
    task_b      = @spawn sum(x)

    all_tasks   = (task_a, task_b)
    all_outputs = fetch.(all_tasks)
end
```

```
julia> @btime maximum($x)
  7.705 ms (0 allocations: 0 bytes)
julia> @btime sum($x)
  3.131 ms (0 allocations: 0 bytes)
julia> @btime multithreaded($x)
  7.741 ms (21 allocations: 1.250 KiB)
```

As we can see, the execution time under multithreaded is roughly equivalent to the maximum time for a single operation to complete in isolation. Nonetheless, this equivalence isn't exact. The reason is that **multithreading has a non-negligible overhead**, stemming from the creation and scheduling of tasks. This determines **multithreading isn't beneficial for operations involving small objects**, as the added overhead negates any potential benefits.

To illustrate this, let's compare the execution times of a sequential and multithreaded approach for different sizes of `x`. In the case considered, the single-threaded approach dominates for sizes smaller than 100,000.

```julia
x_small  = rand(    1_000)
x_medium = rand(  100_000)
x_big    = rand(1_000_000)

function foo(x)
    a           = maximum(x)
    b           = sum(x)

    all_outputs = (a,b)
end
```

```
julia> @btime foo($x_small)
  866.758 ns (0 allocations: 0 bytes)
julia> @btime foo($x_medium)
  59.934 µs (0 allocations: 0 bytes)
julia> @btime foo($x_big)
  620.869 µs (0 allocations: 0 bytes)
```

```julia
x_small  = rand(    1_000)
x_medium = rand(  100_000)
x_big    = rand(1_000_000)

function foo(x)
    task_a      = @spawn maximum(x)
    task_b      = @spawn sum(x)

    all_tasks   = (task_a, task_b)
    all_outputs = fetch.(all_tasks)
end
```

```
julia> @btime foo($x_small)
  3.245 μs (14.33 allocations: 1.068 KiB)
julia> @btime foo($x_medium)
  55.853 μs (21 allocations: 1.250 KiB)
julia> @btime foo($x_big)
  549.445 μs (21 allocations: 1.250 KiB)
```