11c. Task-Based Parallelism: @spawn

Martin Alfaro
PhD in Economics

INTRODUCTION

The previous section introduced the basics of multithreading, highlighting that operations can be computed either sequentially (Julia's default) or concurrently. The latter enables the processing of multiple operations simultaneously, with each of them running as soon as a thread becomes available. This implies that, when Julia's session is initialized with more than one thread, computations can be executed on different CPU cores 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.

Warning!

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. This is why, after grasping a basic understanding of parallelism techniques, we'll devote an entire section on unsafe-thread operations.

ENABLING MULTITHREADING

Each Julia session is initialized with a pool of threads available for execution. Since each thread can run a set of instructions independently, the total number of threads determines how many instructions can be processed simultaneously.

By default, Julia runs in single-threaded mode. To enable multithreading, you need to configure the environment to launch Julia with more than one thread. In VSCode or VSCodium, this can be done by navigating to *File* > *Preferences* > *Settings*. Then, search for the keyword *threads*, which will display the following option:

After selecting *Edit in settings.json*, add the line `"julia.NumThreads": "auto"` in the JSON file that opens. This

setting automatically detects the number of threads supported by your system, typically matching either the logical or physical cores of your CPU.

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

Once a multithreaded session is started, you can perform parallel computations. While several packages exist for this, the current section will focus on the capabilities built directly into Julia. They're provided by the `Threads` package, which is automatically imported in every Julia session.

```
# package Threads automatically imported when you

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 for parallel execution we introduce is the macro `@spawn`. By prefixing an expression with `@spawn`, Julia creates a (non-sticky) task that's immediately scheduled for execution. If a thread is available, the task begins running right away.

Unlike other parallel programming approaches we'll examine later, `@spawn` requires explicit instructions to wait for task completion. The method for doing so depends on the nature of the task output.

The `fetch` function should be employed when tasks produce computational outputs. It serves a dual purpose of waiting for a task to complete and retrieving its return value.Since parallel computation consists of multiple tasks spawned, `fetch` should be broadcasted over a collection containing all the spawned tasks.

The following example demonstrates `fetch` with two spawned tasks, each returning a vector.

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

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

    a,b
end
```

```
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
```

Note that `fetch` takes tasks as its input. Consequently, it's essential to distinguish between `task_a` and `a`:

- `task_a` denotes the task creating the vector `a` (i.e., the task itself),
- `a` refers to the vector created (i.e., the task's output).

For tasks that perform actions but don't return any output, we can use either the function `wait` or the macro `@sync`. The function `wait` works analogously to `fetch`, except that it doesn't return any value. Instead, the macro `@sync` requires

enclosing all operations to be synchronized using the keywords `begin` and `end`.

To illustrate, consider a mutating function. Such functions are suitable as examples, since they modify the contents of a collection in place, without producing a return value.

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

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

```
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
```

```
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 calculate both the sum and the maximum of a vector `x`. Their computation will be implemented following a sequential and a simultaneous approach. To unveil the benefits of parallelization, we'll also include the execution time of each operation in isolation.

The results establish that the total runtime of the sequential procedure is essentially the sum of the individual runtimes. In contrast, the runtime under multithreading is roughly equivalent to the longer of the two computations, thanks to parallelism.

```
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)
```

```
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)
```

Although the multithreaded runtime is close to the maximum of the individual runtimes, the equivalence isn't exact. This is because **multithreading introduces overhead** due to the creation, scheduling, and synchronization of tasks. As a result, **multithreading isn't advantageous for small workloads**, where the overhead can outweigh any potential gains.

To illustrate this effect, 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.

```
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)
```

```
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)
```