11e. Parallel For-Loops

Martin Alfaro
PhD in Economics

INTRODUCTION

For-Loops are fundamental building blocks in scientific computing, often representing the most computationally intensive parts of a program. To accelerate these computations, parallel processing offers a powerful solution by distributing a for-loop's iterations across multiple processor threads. The central challenge, however, lies in how this work is divided, as the optimal strategy depends heavily on the nature of the workload.

This section introduces Julia's `@threads` macro as a straightforward approach to parallelization of for-loops. It operates by dividing the loop iterations evenly among the available threads, before execution begins. For the explanations, we'll contrast its underlying mechanism with the task-based parallelism offered by `@spawn`.

To clearly illustrate the differences between approaches, our examples will use `@spawn` in a simple but inefficient manner,

where a separate task is created for every single iteration. While this is enough for the purpose of comparison with `@threads`, you should bear in mind that it's not representative of typical `@spawn` usage. In the next section, we'll revisit the macro by studying how to efficiently define tasks with `@spawn`. This exploration will reveal that the inherent flexibility of `@spawn` enables the construction of sophisticated parallel strategies, capable of replicating even the behavior of `@threads`.

SOME PRELIMINARIES

Parallelism techniques target code that performs multiple operations, making it a natural fit for for-loops. Using the macro `@spawn` introduced in the previous sections, we can parallelize for-loops through a task-based parallelism. In an upcoming section, we'll demonstrate that `@spawn` is flexible enough to split iterations into tasks in various ways. For now, we'll consider a simple (inefficient) case where each iteration defines a separate task. The coding implementing this technique is shown below.

```
@sync begin
    for i in 1:4
        @spawn println("Iteration $i is computed
    end
end
```

```
Iteration 1 is computed on Thread 1
Iteration 2 is computed on Thread 2
Iteration 4 is computed on Thread 2
Iteration 3 is computed on Thread 2
```

```
@sync begin
    @spawn println("Iteration 1 is computed on Th
    @spawn println("Iteration 2 is computed on Th
    @spawn println("Iteration 3 is computed on Th
    @spawn println("Iteration 4 is computed on Th
end
```

```
Iteration 1 is computed on Thread 1
Iteration 2 is computed on Thread 2
Iteration 3 is computed on Thread 1
Iteration 4 is computed on Thread 2
```

When there are only a few iterations involved in a for-loop, creating one task per iteration can be a straightforward and effective way to parallelize the code. However, as the number of iterations increases, the approach becomes less efficient due to the overhead of task creation. To mitigate this issue, we need to consider alternative ways of parallelizing for-loops.

One such alternative is to create tasks that encompass multiple iterations, rather than just one iteration per task. The

techniques to do this, which will be explored in following sections, offers more granular control, but at the expense of adding substantial complexity to the code.

In light of this complexity, Julia provides the `@threads` macro from the package `Threads`. The goal is to reduce the overhead of task creation, while keeping the parallelization simple. Specifically, `@threads` divides the set of iterations evenly among threads, thereby restricting the creation of tasks to the number of threads available.

The following example demonstrates the implementation of `@threads`, highlighting its difference from the approach with `@spawn`. The scenario considered is based on 4 iterations and 2 worker threads. We also display the thread on which each iteration is executed by using the `threadid()` function, which identifies the thread's ID that's computing the operation.

```
for i in 1:4
    println("Iteration $i is computed on Thread $
end
```

```
Iteration 1 is computed on Thread 1
Iteration 2 is computed on Thread 1
Iteration 3 is computed on Thread 1
Iteration 4 is computed on Thread 1
```

```
@threads for i in 1:4
    println("Iteration $i is computed on Thread $
end
```

```
Iteration 1 is computed on Thread 1
Iteration 2 is computed on Thread 1
Iteration 3 is computed on Thread 2
Iteration 4 is computed on Thread 2
```

```
@sync begin
    for i in 1:4
        @spawn println("Iteration $i is computed
    end
end
```

```
Iteration 2 is computed on Thread 2
Iteration 1 is computed on Thread 1
Iteration 4 is computed on Thread 2
Iteration 3 is computed on Thread 2
```

The key distinction between `@threads` and `@spawn` lies in the strategy for thread allocation. Thread assignments with `@threads` are predetermined: before the for-loop begins, the macro pre-allocates threads and distributes iterations evenly. Thus, each thread is assigned a fixed number of iterations upfront, creating a predictable workload distribution. In the example, the feature is reflected in the allocation of two iterations per thread.

In contrast, `@spawn` creates a separate task for each iteration, dynamically scheduling them as soon as a thread becomes

available. This method allows for more flexible thread utilization, with task assignments adapting in real-time to the current system load and available thread capacity. For instance, in the previous example, a single thread ended up computing three out of the four iterations.

@SPAWN VS @THREADS

The macros `@threads` and `@spawn` embody two distinct approaches to work distribution, thus catering to different types of scenarios. By comparing the creation of one task per iteration relative to `@threads`, we can highlight the inherent trade-offs involved in parallelizing code.

`@threads` employs a coarse-grained approach, making it well-suited for workloads with similar computational requirements. By reducing the overhead associated with task creation, this approach excels in scenarios where tasks have comparable execution times. However, it's less effective in handling workloads with unbalanced execution times, where some iterations are computationally intensive while others are relatively lightweight.

In contrast, `@spawn` adopts a fine-grained approach, treating each iteration as a separate task that can be scheduled independently. This allows for more flexible work distribution,

with tasks dynamically allocated to available threads as soon as they become available. As a result, `@spawn` is particularly well-suited for scenarios with varying computational efforts, where iteration completion times can differ significantly. While this approach has a bigger overhead due to the creation of numerous smaller tasks, it simultaneously enables more efficient resource utilization. This is because no thread remains idle while tasks await computation.

In the following, we demonstrate the efficiency of the approaches under each scenario. With this goal, consider a situation where the *i*-th iteration computes `job(i;time_working)`. This function represents potential calculations performed during `time_working` seconds. It's formally defined as follows.

```
function job(i; time_working)
    println("Iteration $i is on Thread $(threadic

    start_time = time()

    while time() - start_time < time_working
        1 + 1                    # compute `1+1` re
    end
end
```

Note that `job` additionally identifies the thread on which it's running and displays it on the REPL.

Based on a for-loop with four iterations and a session with two worker threads, we next consider two scenarios. They differ by the computational workload of the iterations.

SCENARIO 1: UNBALANCED WORKLOAD

The first scenario represents a situation with unbalanced work, where some iterations require more computational effort. The feature is captured by assuming that the *i*-th iteration has a duration of `i` seconds.

We start by presenting the coding implementing each approach, and then provide explanations for each.

```
function foo(nr_iterations)
    for i in 1:nr_iterations
      job(i; time_working = i)
    end
end
```

```
Iteration 1 is on Thread 1
Iteration 2 is on Thread 1
Iteration 3 is on Thread 1
Iteration 4 is on Thread 1
  10.000 s (40 allocations: 1.562 KiB)
```

```
function foo(nr_iterations)
    @threads for i in 1:nr_iterations
        job(i; time_working = i)
    end
end
```

```
Iteration 1 is on Thread 1
Iteration 3 is on Thread 2
Iteration 2 is on Thread 1
Iteration 4 is on Thread 2
  7.000 s (51 allocations: 2.625 KiB)
```

```
function foo(nr_iterations)
    @sync begin
        for i in 1:nr_iterations
            @spawn job(i; time_working = i)
        end
    end
end
```

```
Iteration 1 is on Thread 1
Iteration 2 is on Thread 2
Iteration 3 is on Thread 1
Iteration 4 is on Thread 2
  6.000 s (69 allocations: 3.922 KiB)
```

Given the execution times for each iteration, a sequential approach would take 10 seconds. As for parallel implementations, `@threads` ensures that there are as many tasks created as number of threads. In the example, this means that there two tasks are created, with the first task computing iterations 1 and 2, and the second task computing

iterations 3 and 4. As a result, the overall execution time is reduced to 7 seconds.

In contrast, `@spawn` creates a separate task for each iteration, which increases the overhead of task creation. Although the overhead is negligible in this example, it can be appreciated in the increased memory allocation. Despite this disadvantage, the approach allows each iteration to be executed as soon as a thread becomes available. Given the varying execution times between iterations, this dynamic allocation becomes advantageous, enabling iterations 3 and 4 to run in parallel.

The example demonstrates this, where iterations 1 and 2 are now executed on different threads. Since the first iteration only requires one second, the thread becomes available to compute the third iteration immediately. The final distribution of tasks on threads is such that iterations 1 and 3 are executed on one thread, while iterations 2 and 4 are executed on the other thread. This results in a total execution time of 6 seconds.

SCENARIO 2: BALANCED WORKLOAD

Consider now a scenario where the execution of `job` requires exactly the same time regardless of the iteration considered. To make the overhead more apparent, we'll use a larger number of iterations. In this context, `@threads` ensures

parallelization with a reduced overhead, explaining why it's faster than the approach relying on `@spawn`.

```
function foo(nr_iterations)
    fixed_time = 1 / 1_000_000

    for i in 1:nr_iterations
      job(i; time_working = fixed_time)
    end
end
```

```
julia> @btime foo(1_000_000)
  1.717 s (without a warmup) (0 allocations: 0 bytes)
```

```
function foo(nr_iterations)
    fixed_time = 1 / 1_000_000

    @threads for i in 1:nr_iterations
        job(i; time_working = fixed_time)
    end
end
```

```
julia> @btime foo(1_000_000)
  858.399 ms (11 allocations: 1.094 KiB, without a warmup)
```

```
function foo(nr_iterations)
    fixed_time = 1 / 1_000_000

    @sync begin
        for i in 1:nr_iterations
            @spawn job(i; time_working = fixed_ti
        end
    end
end
```

```
julia> @btime foo(1_000_000)
  1.270 s (5000021 allocations: 498.063 MiB, 19.16% gc time, withou
```