11f. Parallelization in Practice

Martin Alfaro

PhD in Economics

INTRODUCTION

So far, we've explored two macros for parallelization: `@spawn` and `@threads`. The macro `@spawn` provides granular control over the parallelization process, letting users explicitly define the tasks to be executed concurrently. In contrast, `@threads` offers a simpler approach for parallelizing for-loops, where iterations are automatically partitioned into tasks according to the number of available threads. Furthermore, we've pointed out that, due to inherent dependencies between computations, not all workloads are equally amenable to parallelization. In particular, a naive approach to parallelization can lead to severe issues.

Essentially, our discussions to this point have largely focused on the *syntax* and *work distribution* of parallelization approaches. Therefore, we have yet to address how to apply multithreading in real scenarios. Furthermore, given the possibility of dependencies between computations, *how* to parallelize is only part of the challenge: knowing *when* to parallelize is equally important.

This section and the next one aim to bridge this gap, providing practical guidance on implementing multithreading. We begin by highlighting the advantages of coarse-grained parallelization over fine-grained parallelization. By dividing the workload into a small number of large tasks, coarsegrained parallelization reduces the scheduling overhead from managing numerous lightweight tasks.

After this, we revisit the parallelization of for-loops, this time using `@spawn`. In particular, leveraging the additional control that `@spawn` provides over task creation, we'll demonstrate how to apply multithreading in the presence of a ubiquitous type of dependency: reductions.

We conclude by showing a performance issue arising with multithreading, known as false sharing. While this doesn't affect the correctness the result, it can significantly slow down computations if not addressed.

BETTER TO PARALLELIZE AT THE TOP

Given the overhead involved in multithreading, there's an inherent trade-off between creating new tasks and fully utilizing machine resources. This is why we must always consider whether parallelization is worthwhile in the first place. For instance, when it comes to operations over collections, multithreading is only justified if the collections are large enough to offset the associated overhead. Otherwise, single-threaded approaches will consistently outperform parallelized ones.

In case multithreading is deemed beneficial, we immediately face another decision: at what level code should be parallelized. Next, we'll demonstrate that **parallelism at the highest possible level is preferable to multithreading individual operations**. In this way, we minimize the overhead of task creation.

Note that the level of parallelization is always constrained by the degree of dependency between operations. Hence, our qualification of highest **possible** level. For instance, in problems requiring strictly serial computation, the best we can achieve is parallelization within each individual step.

To illustrate this, let's consider a for-loop where each iteration needs to sequentially compute three operations.

JULIA'S DEFAULT

```julia
step1(a) = a ^ 2
step2(a) = sqrt(a)
step3(a) = log(a + 1)

function all_steps(a)
    y      = step1(a)
    z      = step2(y)
    output = step3(z)

    return output
end

function foo(x)
    output = similar(x)

    for i in eachindex(output)
        output[i] = all_steps(x[i])
    end

    return output
end

x_small  = rand(  1_000)
x_large  = rand(100_000)
```

```julia
julia> @btime foo($x_small)
  5.000 µs (3 allocations: 7.883 KiB)
julia> @btime foo($x_large)
  527.133 µs (3 allocations: 781.320 KiB)
```

PARALLELIZATION AT THE HIGHEST LEVEL POSSIBLE

```julia
step1(a) = a ^ 2
step2(a) = sqrt(a)
step3(a) = log(a + 1)

function all_steps(a)
    y      = step1(a)
    z      = step2(y)
    output = step3(z)

    return output
end

function foo(x)
    output = similar(x)

    @threads for i in eachindex(output)
        output[i] = all_steps(x[i])
    end

    return output
end

x_small  = rand(  1_000)
x_large  = rand(100_000)
```

```julia
julia> @btime foo($x_small)
  11.289 µs (125 allocations: 20.508 KiB)

julia> @btime foo($x_large)
  54.258 µs (125 allocations: 793.945 KiB)
```

```
EACH OPERATION PARALLELIZED

step1(a) = a ^ 2
step2(a) = sqrt(a)
step3(a) = log(a + 1)

function parallel_step(f, x)
    output = similar(x)

    @threads for i in eachindex(output)
        output[i] = f(x[i])
    end

    return output
end

function foo(x)
    y      = parallel_step(step1, x)
    z      = parallel_step(step2, y)
    output = parallel_step(step3, z)

    return output
end

x_small  = rand(  1_000)
x_large  = rand(100_000)
```

```
julia> @btime foo($x_small)
  33.472 µs (375 allocations: 61.523 KiB)
julia> @btime foo($x_big)
  91.834 µs (375 allocations: 2.326 MiB)
```

The examples illustrate the two-step process outlined. First, it shows that parallelization is advantageous only with large collections. Otherwise, the question of whether to parallelize shouldn't even arise. Second, once multithreading proves to be advantageous, it demonstrates that grouping all operations into a single task is faster than parallelizing each operation individually.

IMPLICATIONS

The strategy of parallelizing code at the highest possible level has significant implications for program design, particularly when the program will eventually be applied to multiple independent objects. It suggests a practical guideline: start with an implementation for a single object, without introducing parallelism. After thoroughly optimizing the single-case code, integrate parallel execution at the top level. The approach not only improves performance, but also simplifies the development by making debugging and testing more straightforward.

A common application of this strategy arises in scientific simulations, where independent executions of the same model are required. In such scenarios, the most effective approach is to maintain a single-threaded implementation of the model, while launching multiple instances in parallel. This design ensures that each run remains efficient at the single-thread level and full resource utilization.

THE IMPORTANCE OF WORK DISTRIBUTION

Multithreading performance is heavily influenced by how evenly the computational workload is distributed across iterations. The `@threads` macro is highly effective when each iteration requires roughly equal processing time, as it spawns a task for every iteration. In contrast, scenarios with uneven computational effort can pose significant challenges. In such cases, may finish early and remain idle, while others continue processing heavier tasks. This imbalance undermines parallel efficiency, substantially diminishing the performance gains of multithreading.

To address this issue, we need greater control over how work is distributed among threads. This calls for the use of `@spawn`.

One strategy is to make each iteration a separate task. However, such approach is extremely inefficient if there's a large number of iterations: creating far more tasks than there are threads introduces substantial and unnecessary overhead. The following example illustrates this problem.

@THREADS

```julia
x = rand(10_000_000)

function foo(x)
    output = similar(x)

    @threads for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```julia
julia> @btime foo($x)
  4.907 ms (125 allocations: 76.309 MiB)
```

@SPAWN

```julia
x = rand(10_000_000)

function foo(x)
    output = similar(x)

    @sync for i in eachindex(x)
        @spawn output[i] = log(x[i])
    end

    return output
end
```

```julia
julia> @btime foo($x)
  11.284 s (60005935 allocations: 5.277 GiB)
```

An alternative strategy, which lets users fine-tune the workload of each task, is to partition iterations into smaller subsets that can be processed in parallel. Before detailing the implementation, we'll first explore how to partition a collection and its indices through the `ChunkSplitters` package.

PARTITIONING COLLECTIONS

The package `ChunkSplitters` provides two functions for *lazy* partitioning: `chunks` and `index_chunks`. These functions support `n` and `size` as keyword arguments, depending on the type of partition desired. Specifically, `n` sets the number of subsets to create, with each subset sized to distribute elements evenly. In contrast, `size` specifies the number of elements to be contained in each subset. Since an even distribution across all subsets can't be guaranteed, the package adjusts the number of elements in one of the subsets if necessary.

Below, we apply these functions to a variable `x` that contains the 26 letters of the alphabet. Note that the outputs provided require the use of `collect`, since `chunks` and `index_chunks` are lazy.

PARTITION BY NUMBER OF CHUNKS

```julia
x              = string.('a':'z')           # all letters from "a" to "z"

nr_chunks      = 5

chunk_indices  = index_chunks(x, n = nr_chunks)
chunk_values   = chunks(x, n = nr_chunks)
```

```julia
julia> collect(chunk_indices)
5-element Vector{UnitRange{Int64}}:
 1:6
 7:11
 12:16
 17:21
 22:26

julia> collect(chunk_values)
5-element Vector{SubArray{String, 1, Vector{String}, Tuple{UnitRange{Int64}}, true}}:
 ["a", "b", "c", "d", "e", "f"]
 ["g", "h", "i", "j", "k"]
 ["l", "m", "n", "o", "p"]
 ["q", "r", "s", "t", "u"]
 ["v", "w", "x", "y", "z"]
```

PARTITION BY SIZE OF CHUNKS

```
x               = string.('a':'z')          # all letters from "a" to "z"

chunk_length    = 10

chunk_indices   = index_chunks(x, size = chunk_length)
chunk_values    = chunks(x, size = chunk_length)
```

```
julia> collect(chunk_indices)
3-element Vector{UnitRange{Int64}}:
 1:10
 11:20
 21:26

julia> collect(chunk_values)
3-element Vector{SubArray{String, 1, Vector{String}, Tuple{UnitRange{Int64}}, true}}:
 ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"]
 ["k", "l", "m", "n", "o", "p", "q", "r", "s", "t"]
 ["u", "v", "w", "x", "y", "z"]
```

A relevant partition for multithreading is given by a number of chunks proportional to the number of worker threads. The example below implements this partition, generating both chunk indices and chunk values. Since this partition will eventually be used with for-loops, we also show how to use enumerate to pair each chunk with the values or subindices of its corresponding subset.

PARTITION BY NUMBER OF THREADS

```
x               = string.('a':'z')          # all letters from "a" to "z"

nr_chunks       = nthreads()

chunk_indices   = index_chunks(x, n = nr_chunks)
chunk_values    = chunks(x, n = nr_chunks)
```

```
julia> collect(chunk_indices)
24-element Vector{UnitRange{Int64}}:
 1:2
 3:4
 ⋮
 25:25
 26:26

julia> collect(chunk_values)
24-element Vector{SubArray{String, 1, Vector{String}, Tuple{UnitRange{Int64}}, true}}:
 ["a", "b"]
 ["c", "d"]
 ⋮
 ["y"]
 ["z"]
```

PARTITION BY NUMBER OF THREADS - ENUMERATE

```julia
x               = string.('a':'z')           # all letters from "a" to "z"

nr_chunks       = nthreads()

chunk_indices   = index_chunks(x, n = nr_chunks)
chunk_values    = chunks(x, n = nr_chunks)

chunk_iter1     = enumerate(chunk_indices)    # pairs (i_chunk, chunk_index)
chunk_iter2     = enumerate(chunk_values)     # pairs (i_chunk, chunk_value)
```

```julia
julia> collect(chunk_iter1)
24-element Vector{Tuple{Int64, UnitRange{Int64}}}:
 (1, 1:2)
 (2, 3:4)
 ⋮
 (23, 25:25)
 (24, 26:26)
```

```julia
julia> collect(chunk_iter2)
24-element      Vector{Tuple{Int64,      SubArray{String,      1,      Vector{String},
Tuple{UnitRange{Int64}}, true}}}:
 (1, ["a", "b"])
 (2, ["c", "d"])
 ⋮
 (23, ["y"])
 (24, ["z"])
```

WORK DISTRIBUTION: DEFINING TASKS THROUGH CHUNKS

Leveraging the `ChunkSplitters` package together with `@spawn`, we can control how a for-loop is parallelized by dividing its iterations into well-balanced chunks. Each chunk will then correspond to an independent task to be processed concurrently.

Below, we show how to replicate the behavior of `@threads` via `@spawn`. The two approaches become equivalent when the number of chunks is set equal to the number of worker threads.

@THREADS

```julia
x = rand(10_000_000)

function foo(x)
    output = similar(x)

    @threads for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```julia
julia> @btime foo($x)
  4.907 ms (125 allocations: 76.309 MiB)
```

@SPAWN

```julia
x = rand(10_000_000)

function foo(x, nr_chunks)
    chunk_ranges = index_chunks(x, n=nr_chunks)
    output       = similar(x)

    @sync for chunk in chunk_ranges
        @spawn (@views @. output[chunk] = log(x[chunk]))
    end

    return output
end
```

```julia
julia> @btime foo($x, nthreads())
  5.143 ms (157 allocations: 76.311 MiB)
```

@SPAWN (EQUIVALENT)

```julia
x = rand(10_000_000)

function foo(x, nr_chunks)
    chunk_ranges = index_chunks(x, n=nr_chunks)
    output       = similar(x)
    task_indices = Vector{Task}(undef, nr_chunks)

    for (i, chunk) in enumerate(chunk_ranges)
        task_indices[i] = @spawn (@views @. output[chunk] = log(x[chunk]))
    end

    return wait.(task_indices)
end
```

```julia
julia> @btime foo($x, nthreads())
  4.816 ms (151 allocations: 76.310 MiB)
```

The flexibility of `@spawn` implies we're not constrained to following the approach of `@threads`. For instance, it's common to adopt a partitioning strategy where the number of chunks is proportional to the number of worker threads. This is shown below.

@SPAWN

```julia
x = rand(10_000_000)

function foo(x, nr_chunks)
    chunk_ranges = index_chunks(x, n=nr_chunks)
    output       = similar(x)

    @sync for chunk in chunk_ranges
        @spawn (@views @. output[chunk] = log(x[chunk]))
    end

    return output
end
```

```julia
julia> @btime foo($x, 1 * nthreads())
  5.477 ms (157 allocations: 76.311 MiB)

julia> @btime foo($x, 2 * nthreads())
  8.792 ms (302 allocations: 76.325 MiB)

julia> @btime foo($x, 4 * nthreads())
  6.885 ms (590 allocations: 76.352 MiB)
```

@SPAWN

```julia
x = rand(10_000_000)

function compute!(output, x, chunk)
    @turbo for j in chunk
        output[j] = log(x[j])
    end
end

function foo(x, nr_chunks)
    chunk_ranges = index_chunks(x, n=nr_chunks)
    output       = similar(x)

    @sync for chunk in chunk_ranges
        @spawn compute!(output, x, chunk)
    end

    return output
end
```

```julia
julia> @btime foo($x, 1 * nthreads())
  5.040 ms (133 allocations: 76.310 MiB)

julia> @btime foo($x, 2 * nthreads())
  8.314 ms (254 allocations: 76.323 MiB)

julia> @btime foo($x, 4 * nthreads())
  4.049 ms (494 allocations: 76.347 MiB)
```

HANDLING DEPENDENCIES

So far, our discussion of parallelization has focused on embarrassingly parallel for-loops, where iterations are completely independent. This enables the execution of each iteration in isolation, making parallelization straightforward.

The situation becomes more complex when operations exhibit dependencies. Attempting to parallelize without first addressing these dependencies can lead not only to inefficiencies and wasted resources, but most critically to incorrect results.

There's no one-size-fits-all method for handling dependencies. The appropriate strategy depends on the structure of the specific program. In all cases, though, the approach will require adapting the parallelization technique to work on a reformulated version of the problem. This reformulation must ensure that the tasks to be parallelized are independent—ultimately, parallelization is only possible if independence between operations can be achieved.

Note that, once dependencies are present, some portion of the work will inevitably be non-parallelizable. In fact, no subset of independent tasks may exist at all, as when computations are inherently sequential.

HANDLING REDUCTIONS

A prominent example with dependencies between iterations is reductions. To still benefit from parallelization in this case, the computation must be restructured. The standard approach is to divide the data into chunks, perform partial reductions on each chunk in parallel, and then combine the partial results in a final reduction step. This transformation removes the original dependency between iterations, since each partial reduction now operates on a disjoint subset of the data.

To illustrate, we compute the sum of elements of a vector x . The implementation follows a variant of the partitioning techniques discussed earlier, using `ChunkSplitters` to divide the data into independent segments.

JULIA'S DEFAULT (SEQUENTIAL)

```julia
x = rand(10_000_000)

function foo(x)
    output = 0.

    for i in eachindex(x)
        output += x[i]
    end

    output
end
```

```julia
julia> @btime foo($x)
  5.108 ms (0 allocations: 0 bytes)
```

@THREADS

```julia
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = Vector{Float64}(undef, length(chunk_ranges))

    @threads for (i,chunk) in enumerate(chunk_ranges)
        partial_outputs[i] = sum(@view(x[chunk]))
    end

    return sum(partial_outputs)
end
```

```julia
julia> @btime foo($x)
  1.194 ms (124 allocations: 13.250 KiB)
```

@SPAWN

```julia
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = Vector{Float64}(undef, length(chunk_ranges))

    @sync for (i, chunk) in enumerate(chunk_ranges)
        @spawn partial_outputs[i] = sum(@view(x[chunk]))
    end

    return sum(partial_outputs)
end
```

```julia
julia> @btime foo($x)
  1.169 ms (156 allocations: 13.781 KiB)
```

FALSE SHARING

The issues concerning parallelization up to this point were due to dependencies between operations. Addressing them is necessary, since otherwise we'd obtain incorrect results through a parallelization approach.

Even when those dependencies are removed and therefore correct results are ensured, performance can still be hindered by hardware-level effects. One common bottleneck is **cache contention**, where multiple processor cores compete for access to shared cache resources. One specific manifestation of this issue is **false sharing**, which occurs when multiple cores access data stored in the same cache line. To fully appreciate this issue, it's essential to grasp how CPU caches operate.

Processors use caches to store copies of frequently accessed data. They represent a smaller and faster memory unit than RAM, and are organized into fixed-size blocks called cache lines (typically 64 bytes). When data is needed, the processor first checks the cache. If the data isn't found, this must be

retrieved from RAM and store a copy in the cache, a process that's significantly slower.

When multiple cores access data within the same cache line, the transfer of information is governed by a cache coherency protocol. The goal of this protocol is to ensure consistency across cores. However, the protocol can create inefficiencies: even if one core accesses data that remains unmodified, the presence of altered data within the same cache block may trigger invalidation of the entire line. As a result, all cores are forced to reload the block, despite the absence of a logical need to do so. This phenomenon, known as false sharing, leads to unnecessary cache invalidations and refetches. The outcome is a notable degradation in program performance, particularly in workloads where threads frequently update their variables.

Although false sharing can arise in many multithreaded contexts, it is especially common in reduction operations. This scenario will serve as the focus of our next discussion.

FALSE SHARING IN REDUCTIONS: AN ILLUSTRATION AND SOLUTIONS

Let's consider a simple scenario where the elements of a vector are summed after applying a logarithmic transformation. We'll present two multithreaded implementations to illustrate the impact of false sharing on performance.

The first implementation acts as a baseline and consists of a sequential procedure. The second one suffers from false sharing. The issue arises because multiple threads are repeatedly reading and writing adjacent memory locations in the `partial_outputs` vector. Since CPU cache lines typically span several vector elements, this leads to cache invalidation and forced synchronization between cores.

SEQUENTIAL

```julia
x = rand(10_000_000)

function foo(x)
    output = 0.

    for i in eachindex(x)
        output += log(x[i])
    end

    output
end
```

```julia
julia> @btime foo($x)
  35.349 ms (0 allocations: 0 bytes)
```

FALSE SHARING

```julia
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = zeros(length(chunk_ranges))

    @threads for (i,chunk) in enumerate(chunk_ranges)
        for j in chunk
            partial_outputs[i] += log(x[j])
        end
    end

    return sum(partial_outputs)
end
```

```julia
julia> @btime foo($x)
  14.018 ms (124 allocations: 13.250 KiB)
```

There are several strategies to address false sharing. All of them base its strategy on preventing threads from repeatedly accessing the same cache line.

Considering why false sharing arises, an intuitive way to handle it is by **vector padding**, where we add extra spacing between elements in memory. By ensuring that each thread's accumulator is placed on a distinct cache line, concurrent updates no longer interfere with one another at the cache level. In practice, this can be implemented by storing partial outputs in a matrix with sufficient separation between rows. Below, we do this by inserting seven empty rows between each partial output.

PADDING

```julia
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = zeros(7, length(chunk_ranges))

    @threads for (i,chunk) in enumerate(chunk_ranges)
        for j in chunk
            partial_outputs[1,i] += log(x[j])
        end
    end

    return sum(@view(partial_outputs[:,1]))
end
```

```julia
julia> @btime foo($x)
  3.843 ms (124 allocations: 14.500 KiB)
```

While intuitive, the solution lacks practical appeal. Consequently, we introduce more common approaches to addressing false sharing. The first two presented below introduce a thread-local variable `temp` to store partial results. In this way, each thread maintains its own accumulator, writing to the shared array only once at the end. We implement this solution via `@threads` and `@spawn`. After

that, we present an alternative solution where each partial reduction is computed through a separate function. This addresses false sharing in a similar spirit, where accumulation is based on a variable that's local to a function.

LOCAL VARIABLE (@THREADS)

```julia
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = zeros(length(chunk_ranges))

    @threads for (i,chunk) in enumerate(chunk_ranges)
        temp = 0.0
        for j in chunk
            temp += log(x[j])
        end
        partial_outputs[i] = temp
    end

    return sum(partial_outputs)
end
```

```julia
julia> @btime foo($x)
  3.621 ms (124 allocations: 13.250 KiB)
```

LOCAL VARIABLE (@SPAWN)

```julia
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = zeros(length(chunk_ranges))

    @sync for (i,chunk) in enumerate(chunk_ranges)
        @spawn begin
            temp = 0.0
            for j in chunk
                temp += log(x[j])
            end
            partial_outputs[i] = temp
        end
    end

    return sum(partial_outputs)
end
```

```julia
julia> @btime foo($x)
  3.407 ms (156 allocations: 13.781 KiB)
```