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 have largely focused on the *syntax* and *work distribution* of parallelization approaches. Yet, we have 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, coarse-grained 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 have enough elements 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, compared to multithreading individual operations**. By adopting this strategy, 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, let's consider a for-loop where each iteration needs to sequentially compute three operations.

JULIA'S DEFAULT

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

PARALLELIZATION AT THE HIGHEST LEVEL POSSIBLE

```
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> @btime foo($x_small)
  13.667 μs (125 allocations: 20.508 KiB)
julia> @btime foo($x_large)
  71.050 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($x_small)
  35.841 µs (375 allocations: 61.523 KiB)
julia> @btime foo($x_big)
  104.260 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. In particular, 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. In those cases, independent executions of the same model are required. Thus, the most effective approach is to maintain a single-threaded implementation of the model, eventually launching multiple instances in parallel. This design ensures that each run remains efficient at the single-thread level, while taking advantage of 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 spawns a task for every iteration, making it highly effective when each iteration requires roughly equal processing time. In contrast, scenarios with uneven computational effort are more challenging. In such cases, some threads 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

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

@SPAWN

```
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> @btime foo($x)
  11.095 s (60005888 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 along with 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 in all cases, 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

```
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> 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"]
```

For multithreading, a relevant partition is a number of chunks proportional to the number of worker threads. The example below implements this, 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

```
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> collect(chunk_iter1)
24-element Vector{Tuple{Int64, UnitRange{Int64}}}:
 (1, 1:2)
 (2, 3:4)
 ⋮
 (23, 25:25)
 (24, 26:26)
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 parallelized. The process divides iterations into chunks, where each of them correspond to a independent task to be processed concurrently.

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

@THREADS

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

@SPAWN

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

@SPAWN (EQUIVALENT)

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

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

@SPAWN

```
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> @btime foo($x, 1 * nthreads())
  7.058 ms (157 allocations: 76.311 MiB)
julia> @btime foo($x, 2 * nthreads())
  5.492 ms (302 allocations: 76.325 MiB)
julia> @btime foo($x, 4 * nthreads())
  4.982 ms (590 allocations: 76.352 MiB)
```

@SPAWN

```
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> @btime foo($x, 1 * nthreads())
  4.379 ms (133 allocations: 76.310 MiB)
julia> @btime foo($x, 2 * nthreads())
  4.529 ms (254 allocations: 76.323 MiB)
julia> @btime foo($x, 4 * nthreads())
  4.080 ms (494 allocations: 76.347 MiB)
```

Having more chunks than number of threads can make sense when, for instance, the processor cores don't offer uniform performance. When this is the case, the strategy will ensure that there th fastest cores don't remain idle, so that there's full resource utilization of computational resources.

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 common technique that exhibits dependency between iterations is reductions. To still benefit from parallelization despite this dependency, its computation must be reorganized. The typical strategy is to partition the data into chunks, perform a partial reduction on each chunk in parallel, and then combine the partial results in a final reduction step. This restructuring removes the original dependency between iterations, since each partial reduction now operates on a disjoint subset of the data.

To illustrate, let's compute the sum of elements of a vector `x`. The implementation relies on the `ChunkSplitters` package to divide the data into independent segments.

JULIA'S DEFAULT (SEQUENTIAL)

```
x = rand(10_000_000)

function foo(x)
    output = 0.0

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

    output
end
```

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

@THREADS

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

@SPAWN

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

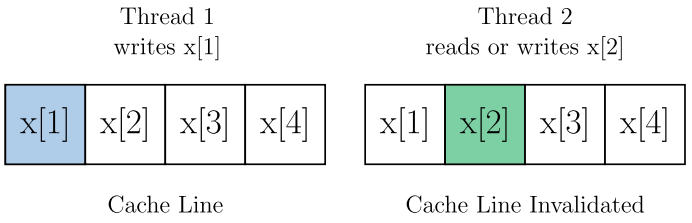
FALSE SHARING

Addressing dependency between operations is necessary when parallelizing, since the program may otherwise yield incorrect results. However, even after removing dependencies and ensuring correctness, performance can still suffer due to hardware-level effects.

One common bottleneck is **cache contention**, where multiple processor cores compete for shared cache resources. A manifestation of this issue is what's known as **false sharing**, where multiple cores simultaneously access data stored in the same cache line. To understand why this degrades performance, it helps to first review how CPU caches work.

Processors rely on caches to hold copies of frequently accessed data. These caches are much smaller but significantly faster than main memory (RAM), and they're organized into fixed-size blocks called cache lines (typically 64 bytes). When the processor needs a piece of data, it first checks whether it's already present in the cache. If not, the data must be fetched from RAM and placed into a cache line, a process that's considerably slower.

Likewise, when multiple cores access data within the same cache line, the transfer of information is governed by a cache coherency protocol. Its goal is to ensure consistency across cores. However, the protocol can create inefficiencies: even if one core accesses data that remains unmodified, any modification to another value within the same cache line may cause the entire line to be invalidated. 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.



Below, we focus on the emergence of false sharing in reduction operations.

FALSE SHARING IN REDUCTIONS: AN ILLUSTRATION AND SOLUTIONS

Suppose that the elements of a vector are summed after applying a logarithmic transformation. We'll present two implementations. The first one acts as a baseline and consists of a sequential procedure. The second one is multithreaded but suffers from false sharing. The problem arises because each thread writes its partial sum into a different element of the `partial_outputs` vector. Although these elements are logically independent, they're stored in adjacent memory locations that fall within the same cache line. As a result, every update by one thread invalidates the cache line in other cores, forcing unnecessary reloads that degrade performance.

SEQUENTIAL

```
x = rand(10_000_000)

function foo(x)
    output = 0.0

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

    output
end
```

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

FALSE SHARING

```
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)
        for j in chunk
            partial_outputs[i] += log(x[j])
        end
    end

    return sum(partial_outputs)
end
```

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

There are several strategies to address the issue. All of them ensure that threads don't write to data residing in the same cache line.

One common strategy is **vector padding**, where extra spacing is inserted between the elements of `partial_outputs`. This strategy ensures that each thread's accumulator is placed on a distinct cache line, so that concurrent writes no longer interfere with one another at the cache level. We implement this below by storing the partial outputs in a vector with sufficient separation between rows. In particular, a separation of 8 entries.

PADDING

```
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    nr_strides      = 8
    partial_outputs = Vector{Float64}(undef, length(chunk_ranges) * nr_strides)

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

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

```
julia> @btime foo($x)
  6.243 ms (124 allocations: 14.625 KiB)
```

While intuitive, the solution lacks practical appeal. In fact, the exact number of rows to insert depends on the element type and the underlying computer architecture. For this reason, let's introduce more efficient general approaches to avoiding false sharing.

The first method introduces a thread-local variable `temp` to store partial results. In this way, each thread updates only its own local variable, finally writing to the shared array exactly 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 inside a separate function. This addresses false sharing in a similar spirit, where accumulation is based on a local variable.

LOCAL VARIABLE (@THREADS)

```
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)
        temp = 0.0
        for j in chunk
            temp += log(x[j])
        end
        partial_outputs[i] = temp
    end

    return sum(partial_outputs)
end
```

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

LOCAL VARIABLE (@SPAWN)

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

FUNCTION

```
x = rand(10_000_000)

function compute(x, chunk)
    temp = 0.0

    for j in chunk
        temp += log(x[j])
    end

    return temp
end

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] = compute(x, chunk)
    end

    return sum(partial_outputs)
end
```

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