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. Focusing on programs that aren't embarrassingly parallel, we've demonstrated that a naive approach to parallelization can lead to severe issues.

Overall, our discussions to this point have largely focused on the *syntax* and *work distribution* of these 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, 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 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 parallelizing our code 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

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

PARALLELIZATION AT THE HIGHEST LEVEL POSSIBLE

```
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($x_small)
  11.289 μs (125 allocations: 20.508 KiB)
julia> @btime foo($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 is deemed beneficial, 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 by implementing the code for a single object, without introducing parallelism. After thoroughly optimizing the single-case code, parallel execution can then be seamlessly integrated at the top level. The approach not only improves performance, but also simplifies the development by making debugging and testing more straightforward.

A common example of this strategy appears in scientific simulations, where many independent runs of the same model must be executed. In such cases, the most effective method is to maintain a single-threaded codebase for the model itself, and then run multiple instances of that model in parallel. This ensures that each run is as efficient as possible, while still taking full advantage of available computing resources.

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, since the tasks it spawns will contain an equal amount of iterations. However, scenarios with uneven computational effort can pose significant challenges. In such cases, some threads may remain idle while others are heavily loaded, dramatically reducing the potential performance gains of parallel processing.

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, this approach becomes 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 effect, showing that spawning one task per iteration can result in extremely slow execution times.

@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)
  4.907 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)
  11.284 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. This step makes use of the `ChunkSplitters` package.

PARTITIONING COLLECTIONS

The package `ChunkSplitters` makes it possible to partition a collection `x` and its indices. It 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` containing the 26 letters of the alphabet. Note that outputs 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"]
```

A relevant type of partition for multithreading is given by a number of chunks proportional to the number of worker threads. The example below implements this partition. It generates both chunk indices and chunk values.

Since this partition approach 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

After covering how to partition datasets with the `ChunkSplitters` package, we can now turn to its role in multithreading. By dividing iterations into well-balanced chunks, we can assign each subset as an independent task to be processed concurrently.

Below, we parallelize a for-loop with `@threads`. Then, we present an equivalent implementation with `@spawn`. This requires defining a number of chunks that equals 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)
  4.907 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())
  5.143 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())
  4.816 ms (151 allocations: 76.310 MiB)
```

Unlike `@threads`, the `@spawn` macro offers finer control over how tasks are allocated to threads. This flexibility means we're not constrained to replicate the `@threads`'s behavior. For example, we could adopt different partitioning strategies where the number of chunks is proportional to the number of worker threads, as shown in the following examples.

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

```
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($x, 1 * nthreads())
  5.040 ms (133 allocations: 76.310 MiB)
julia> @btime foo($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. In such cases, each iteration can be executed in isolation, making parallelization straightforward.

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

The issue occurs because 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 of operations can be achieved.

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

HANDLING REDUCTIONS

A prominent example where dependencies between iterations arise is in reductions. To address these dependencies and still benefit from parallelization, 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, because each partial reduction operates on a disjoint subset of the data.

To illustrate, let's 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)

```
x = rand(10_000_000)

function foo(x)
    output = 0.

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

    output
end
```

```
julia> @btime foo($x)
  5.108 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)
  1.194 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)
  1.169 ms (156 allocations: 13.781 KiB)
```

FALSE SHARING

So far, we've focused on constraints on parallelization due to dependencies between operations. But even when those dependencies are eliminated and the parallelized units are logically independent, performance can still be limited by hardware-level effects. One common culprit is cache contention, where multiple processor cores compete for shared cache resources. A particular manifestation of this issue, known as **false sharing**, occurs when multiple cores access data stored in the same cache line. Understanding this issue requires grasping how CPU caches function.

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 data follows a cache coherency protocol. This is designed to maintain data consistency across cores. This protocol can lead to situations where one core accesses data that isn't modified by another core, yet shares a cache block with altered data. In such cases, the entire cache line may be invalidated, forcing the cores to reload the entire cache block, despite there being no logical necessity to do so. This phenomenon is known as false sharing, and can cause unnecessary cache invalidations and refetches. The consequence is a significant degradation of the program's performance, particularly if threads frequently modify their variables.

While false sharing can occur in various multithreading scenarios, it's particularly prevalent in reduction operations. This case will be our focus next.

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 is a naive approach that closely resembles a typical sequential implementation. Its goal is to illustrate 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.

In contrast, the second implementation avoids false sharing, and we'll analyze why this is so after presenting the code snippets.

SEQUENTIAL

```
x = rand(10_000_000)

function foo(x)
    output = 0.

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

    output
end
```

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

FALSE SHARING

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

LOCAL VARIABLE (@THREADS)

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

To address false sharing in parallel reductions, there are several strategies that can be employed. All of them aim to prevent threads from repeatedly accessing the same cache line.

The previous example already presented one solution. It involves introducing a thread-local variable called `temp` to accumulate results. In this way, each thread maintains its own accumulator, writing to the shared array only once at the end.

Two additional solutions are presented below. The first one entails computing the reduction through a separate function. This addresses false sharing by the same logic as before, where the accumulation is done through a variable local to a function. The second solution involves defining `partial_outputs` as a matrix with extra rows (seven in particular), a technique known as vector padding. This approach guarantees that each thread's accumulator is allocated on a different cache line, so that concurrent updates don't interfere with each other at the cache level.

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 = zeros(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)
  3.623 ms (124 allocations: 13.250 KiB)
```

PADDING

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