11f. Applying Parallelization

Martin Alfaro
PhD in Economics

INTRODUCTION

So far, we've explored two approaches for parallelizing code, tailored to different scenarios. The first one was `@spawn`. By allowing us to define the specific tasks to be processed, this provides granular control over the parallelization process. For its part, `@threads` represents a simplified approach to parallelizing for-loops, where the tasks spawned are automatically handled. In particular, they're defined based on the number of available threads.

Additionally, we've pointed out that, due to inherent dependencies between computations, not all tasks lend themselves equally to parallelization. Specifically, when tasks aren't embarrassingly parallel, a naive approach can lead to severe issues like race conditions.

All this implies that, to this point, our discussion has been exclusively focused on the syntax and work distribution of these approaches. Therefore, we have yet to address how to apply multithreading in real scenarios, including strategies to deal with dependencies.

This section and the next one aim to bridge this gap, providing practical guidance on implementing multithreading. With this goal, we begin by showing the advantages of parallelizing at a coarse level compared to parallelization at individual operations. After this, we introduce a more general method based on `@spawn` to parallelize for-loops. This enables us to have control over the partition of iterations that define tasks. Its main advantage is the provision of a flexible distribution of iterations among tasks, thus giving us further control over task creation. The technique also makes it possible to apply multithreading under a ubiquitous type of dependency: reductions.

BETTER TO PARALLELIZE AT THE TOP

Given the overhead involved in multithreading, there's an inherent trade off between creating new tasks and utilizing all our machine resources. Consequently, we must first carefully consider whether parallelizing our code is worthy. For instance, multithreading is only justified with collections if their sizes are substantial enough to make up for overhead involved. Otherwise, single-threaded approaches will consistently outperform parallelized ones.

In case multithreading is worthwhile, we immediately face another decision: at what level to parallelize code. In the following, we'll demonstrate that **parallelism at the highest level possible is preferable to multithreading individual operations**. The reason is that the former minimizes the overhead involved in creating tasks, thus resulting in faster execution times.

Note that the level of parallelization is limited by the degree of dependence between operations, explaining why we qualify the highest level as the one that's *possible*. For instance, in problems requiring serial computation, the best we can achieve is a parallelization at each individual step.

To illustrate all 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> @btime foo($x_small)
  4.923 μs (1 allocations: 7.938 KiB)
julia> @btime foo($x_large)
  507.797 μs (2 allocations: 781.297 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> @btime foo($x_small)
  9.963 μs (122 allocations: 20.547 KiB)
julia> @btime foo($x_large)
  59.551 μs (123 allocations: 793.906 KiB)
```

EACH OPERATION PARALLELIZED

```julia
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($x_small)
  35.938 μs (366 allocations: 61.641 KiB)
julia> @btime foo($x_big)
  87.746 μs (369 allocations: 2.326 MiB)
```

The example clearly illustrates that parallelization is only advantageous when dealing with big collections. This is evidenced by the execution times, where multithreading is only faster when `x_large` is considered. Secondly, an approach where tasks comprise all operations is faster than multithreading each in isolation. This is in part reflected in the reduced memory allocations when all operations are encompassed.

IMPLICATIONS

The strategy of parallelizing code at the top level has significant implications for writing programs. This is especially the case when the code will eventually be applied to multiple objects. It suggests that we should start by writing code for a single object, without considering parallelization. Once the single-case code is thoroughly optimized, parallel execution can be seamlessly integrated at the highest level. The approach not only improves performance, but also simplifies the coding process by streamlining the debugging and testing of code.

A typical example where this strategy arises naturally is scientific simulations, where numerous independent runs of the same model are executed. In this case, the best strategy is to focus on a single-thread codebase for the model, subsequently processing the different runs of the model in parallel.

THE IMPORTANCE OF WORK DISTRIBUTION

Multithreading performance is influenced by the balance of computational workload across iterations. The `@threads` macro is highly effective when each iteration requires a roughly equal processing time. This is because the tasks spawned will comprise an equal amount of iterations. However, scenarios with varying computational effort can pose significant challenges. In those cases, some threads may remain idle, while others will be heavily loaded. This may dramatically reduce the potential performance gains of parallel processing.

To address this issue, we need to have more control over how to distribute work among threads. With the tools introduced so far, we could employ `@spawn` to make each iteration represent a different task. However, this approach becomes extremely inefficient if there is a substantial number of iterations. The reason is that defining many more tasks than the number of threads available results in an unnecessary substantial overhead. The following example reveals this feature, where the execution time of spawning one task per iteration is extremely slow.

@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> @btime foo($x)
  4.942 ms (123 allocations: 76.306 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> @btime foo($x)
  9.983 s (60001697 allocations: 5.136 GiB)
```

To have more control over the work distribution, we need to partition objects into smaller subsets that can be processed concurrently. Before explaining the implementation of the technique, we begin showing how to partition a collection and its indices. The procedure relies on the `ChunkSplitters` package.

PARTITIONING COLLECTIONS

The easiest way to partition a collection `x` and its indices `x` is through the package `ChunkSplitters`. The package provides two functions for *lazily* partitioning called `chunks` and `index_chunks`. The functions accept `n` and `size` as keyword arguments, depending on the partition to be implemented. Specifically, `n` specifies the number of subsets in which the collection should be divided, where each subset's size attempts to distribute elements evenly. In contrast, `size` provides the number of elements that each subset should contain. Since `size` can't guarantee an even distribution across all subsets, it'll adjust the number of elements in one of the subsets.

The following example considers a variable `x` that comprises the 26 letters of the alphabet. Note that the presentation of the outputs uses `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> 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"]
```

One common way to apply partitions for multithreading is by considering a number of chunks that are proportional to the number of worker threads. Moreover, iterations can be based on `enumerate` to get pairs of chunk index and either the subcollection or subindices.

PARTITION BY NUMBER OF THREADS

```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_iter     = enumerate(chunk_indices)    # pairs (i_chunk, chunk_index)
```

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

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

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

WORK DISTRIBUTION: DEFINING TASKS THROUGH CHUNKS

To define tasks through chunks, we need to partition the collection into smaller subsets that can be processed concurrently. We've already discussed how to apply these techniques using the `ChunkSplitters` package.

Let's consider a simple example where we want to parallelize a for-loop:

@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> @btime foo($x)
  4.942 ms (123 allocations: 76.306 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> @btime foo($x)
  5.301 ms (156 allocations: 76.308 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> @btime foo($x)
  4.893 ms (148 allocations: 76.307 MiB)
```

The approach provides more control over the allocation of tasks to threads. For instance, we could define the number of chunks as proportional to the number of worker threads.

@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> @btime foo($x, 1 * nthreads())
  4.923 ms (156 allocations: 76.308 MiB)
julia> @btime foo($x, 2 * nthreads())
  4.898 ms (301 allocations: 76.323 MiB)
julia> @btime foo($x, 4 * nthreads())
  4.462 ms (589 allocations: 76.349 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> @btime foo($x, 1 * nthreads())
  4.418 ms (132 allocations: 76.307 MiB)
julia> @btime foo($x, 2 * nthreads())
  4.546 ms (253 allocations: 76.320 MiB)
julia> @btime foo($x, 4 * nthreads())
  3.659 ms (493 allocations: 76.344 MiB)
```

PARALLEL REDUCTIONS

So far, our exploration of parallelization has focused on cases with independent tasks. In particular, the iterations in for-loops were independent, thereby defining an embarrassingly parallel program. This was a deliberate choice, as not all tasks lend themselves to parallelization, due to inherent dependencies between computations. In particular, when tasks aren't embarrassingly parallel, a naive approach for their computation can not only lead to inefficiencies, but actually introduce critical issues, including incorrect results.

Nonetheless, depending on the nature of the dependency, we can adapt our parallelization strategy to still benefit from parallelization. One strategy involves dividing a large task into *smaller independent sub-tasks* that can be executed concurrently. By doing so, we can execute the subtasks in parallel without compromising the correctness of the results, as each sub-task remains independent of the others. Once all sub-tasks complete, their results are combined to generate the final output.

The approach is particularly suitable for reduction operations. Moreover, its technical implementation is a variant of the partition techniques previously presented. To illustrate it, we consider the simplest scenario possible, where we simply compute the sum of elements of a vector `x`.

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> @btime foo($x)
  4.909 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> @btime foo($x)
  1.163 ms (122 allocations: 13.234 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> @btime foo($x)
  1.163 ms (155 allocations: 13.750 KiB)
```

FALSE SHARING IN REDUCTIONS

Cache contention represents a performance challenge where multiple processor cores compete for shared cache resources. A particular manifestation of this issue known as **false sharing** arises when multiple cores access data stored in the same cache line. To understand this issue, it's essential to grasp 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.

AN ILLUSTRATION AND SOLUTIONS FOR REDUCTIONS

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

```julia
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)
  33.629 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> @btime foo($x)
  12.579 ms (122 allocations: 13.234 KiB)
```

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> @btime foo($x)
  3.579 ms (122 allocations: 13.234 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> @btime foo($x)
  3.379 ms (155 allocations: 13.750 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 address 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 that concurrent updates don't interfere with each other at the cache level.