11f. Parallelization in Practice

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 analyze 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. 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`, possibly deploying different strategies.

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 is parallelized. Instead of relying on the fixed scheduling strategy of `@threads`, the package explicitly divides the iteration space into user-defined chunks, with each chunk mapped to an independent task executed concurrently.

Below, we illustrate possible strategies to define chunks. To begin with, we replicate the exact behavior of `@threads` via `@spawn`. Specifically, both approaches follow the same execution pattern when the number of chunks matches 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)
```

However, the flexibility of `@spawn` means we're not limited to the partitioning scheme used by `@threads`. A widely used strategy is to create more chunks than threads to improve load balancing. This is especially effective on systems where cores don't offer uniform performance. By using smaller chunks, faster cores can pick up additional work as soon as they finish their current tasks, preventing idle time and therefore fully utilizing the available hardware.

The next example demonstrates this approach by choosing numbers of chunks proportional to the number of worker threads.

@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)
```

HANDLING DEPENDENCIES

So far, our discussion of parallelization has focused on embarrassingly parallel for-loops, where each iteration is completely independent of the others. In these situations, every iteration can run in isolation, which makes parallelization conceptually simple and very effective.

Things become more subtle once dependencies enter the picture. When operations rely on the results of earlier computations, attemps to parallelization without first addressing those dependencies can lead to wasted work, poor performance, or even incorrect results.

There's no universal recipe for handling dependencies, because the right approach depends entirely on the structure of the program. In practice, you need to reformulate the computation so that the units of work you intend to parallelize are independent. Only after this restructuring can parallelization proceed safely. If such a reformulation isn't possible, then parallelization simply isn't viable for that part of the computation.

It's also important to recognize that once dependencies are present, some fraction of the work will inevitably remain sequential. In extreme cases, the computation may be inherently serial, leaving no subset of independent tasks to parallelize at all.

REDUCTIONS AS A PARTICULAR TYPE OF DEPENDENCE

Reductions are a common technique that introduces dependencies between iterations. To take advantage of parallel execution despite this dependency, the computation must be reorganized. The standard approach is to partition the data into chunks, perform a partial reduction on each chunk in parallel, and then combine those partial results in a final reduction step. This restructuring removes the original loop-carried dependency, since each partial reduction operates on a disjoint subset of the data and therefore doesn't interfere with the others.

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)
```

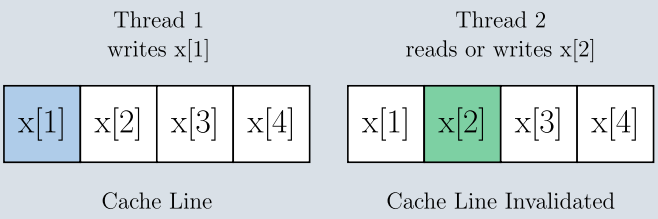
FALSE SHARING

Addressing dependencies between operations is essential when parallelizing; without doing so, a program can easily produce incorrect results. Yet even after eliminating these dependencies and ensuring correctness, performance may still degrade due to underlying 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

Consider the task of summing the elements of a vector after applying a logarithmic transformation to each entry. To illustrate how implementation choices affect performance, we'll look at two versions of this computation. The first is a straightforward sequential routine that processes the vector from start to finish. It serves as a clear baseline: simple, predictable, and free of concurrency concerns.

The second implementation parallelizes the work by assigning different segments of the vector to different threads, each responsible for accumulating its own partial sum. At first glance, this design seems efficient, since the partial sums are logically independent. However, it introduces a subtle performance pitfall: false sharing.

In this version, each thread stores its partial result in a distinct element of the `partial_outputs` array. Even though these elements represent independent data, they're typically placed in contiguous memory locations. When several of these locations fall within the same cache line, updates from one thread cause that entire line to be invalidated on other cores. Those cores must then reload the line before continuing, creating unnecessary coherence traffic. This repeated invalidation and reloading can significantly slow down the computation, despite the algorithm's apparent parallel structure.

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)
```

Several techniques can mitigate this problem, all of which aim to prevent multiple threads from writing to memory locations that share a 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)
```

Although padding is intuitive, it's not especially practical. The amount of spacing required depends on both the element type and the machine's cache-line size, which varies across architectures. Rather than tuning these details manually, it's usually better to rely on approaches that avoid false sharing in a more robust and portable way.

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`.

An alternative solution is to compute each partial reduction inside a separate function. Because the function's local variables are thread-private, the accumulation proceeds without any shared writes until the final store.

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)
```