11g. Multithreading Packages

Martin Alfaro
PhD in Economics

INTRODUCTION

Parallelizing code may seem straightforward at a first glance. However, once we start delving into its subtleties, it's rapidly revealed that an effective implementation can be a daunting task. As we've discussed, naive implementations can lead to various issues, including performance problems like suboptimal load balancing or false sharing, and more severe concerns such as data races. Furthermore, even if the necessary skills for a correct implementation were mastered, the added complexity can severely impair the code's readability and maintainability.

To assist users in overcoming these obstacles, several packages for parallelization have emerged. These tools aim to simplify the implementation of multithreading, allowing users to leverage its benefits without grappling with low-level intricacies. In this section, we'll present a few of these packages. In particular, the focus will be on those that facilitate the application of multithreading to embarrassingly parallel problems and reductions.

The first package we explore is `OhMyThreads`. This offers a collection of high-level functions and macros that help developers parallelize operations with minimal effort. For instance, it eliminates the need to manually partition tasks and tackles subtle performance issues like false sharing. We'll then examine the `Polyester` package. Thanks to its reduced overhead, this package is capable of streamlining parallelization for small objects. After this, we revisit the `LoopVectorization` package. In particular, we introduce the macro `@tturbo`, which combines the benefits of SIMD instructions with multithreading.

PACKAGE "OHMYTHREADS"

As the package's documentation claims, `OhMyThreads` aims to be a user-friendly package to seamlessly apply multithreading. Consistent with its minimalist approach, the package only introduces a handful of essential functionalities that could easily be part of Julia's `Base`. The goal is to allow users to implement code parallelization, even if they don't possess deep expertise in the subject.

Specifically, the package provides various higher-order functions and macros that internally handle data-race conditions and performance issues like false sharing. Despite its simplicity, `OhMyThreads` still covers a significant range of scenarios, even supporting reduction operations.

In the following, we present the main higher-functions provided by the package. Before introducing them, let's indicate several features that these functions share. Firstly, their names in `OhMyThreads` mirror those in `Base`, but adding a prefix of `t`. For instance, the counterpart to `map` is `tmap`. Secondly, `OhMyThreads` offers the option of customizing the parallelization process. This is achieved

through an integration with `ChunkSplitters`, enabling customized work distributions among tasks via two keyword arguments: `nchunks`(or equivalently `ntasks`) to define the number of subsets in the partition, and `chunksize` to specify the number of elements in each subset.

> **Warning!**
> All the code snippets below assume you've already loaded the package with `using OhMyThreads`.

PARALLEL MAPPING

The `tmap` function serves as the multithreaded counterpart to `map`. Its syntax is `tmap(foo, x)`, where `foo` is the transforming function applied to each element of the collection `x`. Unfortunately, applying `tmap` in this form results in a performance loss. This is due to a technicality, arising from the type instability of the object `Task`.

To circumvent this issue and regain the lost performance, we must then explicitly indicate the output's type. To do this, we need the function method `tmap(foo, T , x)`, where `T` represents the *element type* of the output. Thus, if for instance the output is a `Vector{Float64}`, `T` would be `Float64`. Instead of directly declaring `T`, a more flexible alternative is to use `eltype(x)`, making the output's type mirror that of `x`.

The package also provides an in-place version, `tmap!`. In this case, since `tmap!` requires specifying the output vector, there's no need to do any extra work to avoid performance losses.

Below, we illustrate the application of these functions. To provide a basis for comparison, we include results of `map` and `map!` as single-threaded baselines.

```
x                    = rand(1_000_000)


foo(x)               =  map(log, x)
foo_parallel1(x)     = tmap(log, x)
foo_parallel2(x)     = tmap(log, eltype(x), x)
```

```
julia> foo($x)
  3.254 ms (2 allocations: 7.629 MiB)
julia> foo_parallel1($x)
  1.494 ms (568 allocations: 16.958 MiB)
julia> foo_parallel2($x)
  337.724 μs (155 allocations: 7.642 MiB)
```

```
x                       = rand(1_000_000)
output                  = similar(x)

foo!(output,x)          =  map!(log, output, x)
foo_parallel!(output,x) = tmap!(log, output, x)
```

```
julia> foo($x)
  3.303 ms (0 allocations: 0 bytes)
julia> foo_parallel!($x)
  334.747 μs (150 allocations: 13.188 KiB)
```

`OhMyThreads` additionally provides the option to control the work distribution among tasks. This is done through the keyword arguments `nchunks` and `chunksize`, which are internally implemented via the package `ChunkSplitters`. Specifically, `nchunks` controls the number of subsets in the partition, while `chunksize` sets the number of elements per task. Note that `nchunks` and `chunksize` are mutually exclusive options, so that only one of them can be used at a time.

To illustrate the use `nchunks`, we'll set its value equal to `nthreads()`. By setting a number of chunks equal to the number of worker threads, we're adopting an even distribution among tasks, similar to how `@threads` operates. To set the same number with `chunksize`, we'll make use of the floor division operator `÷`. This is a binary operator that rounds a division down to the nearest integer towards zero. [1]

```
x      = rand(1_000_000)

foo(x) = tmap(log, eltype(x), x; nchunks = nthreads())
```

```
julia> @btime foo($x)
  339.006 μs (155 allocations: 7.642 MiB)
```

```
x      = rand(1_000_000)

foo(x) = tmap(log, eltype(x), x; chunksize = length(x) ÷ nthreads())
```

```
julia> @btime foo($x)
  355.825 μs (164 allocations: 7.643 MiB)
```

> **Do-Block Syntax**
>
> When `tmap` requires passing more complex functions, we can still use an anonymous function. In this case, the do-block syntax comes in handy. It enables the creation of multi-line functions, making code more readable. Below, we show an example.

```
x = rand(1_000_000)

function foo(x)

    output = tmap(a -> 2 * log(a), x)



    return output
end
```

```
x = rand(1_000_000)

function foo(x)

    output = tmap(x) do a
                2 * log(a)
             end

    return output
end
```

ARRAY COMPREHENSIONS

`OhMyThreads` also provides an alternative to `tmap` via array comprehensions. Unlike the standard implementation in `Base`, the version from `OhMyThreads` combines a multithreaded variant of `collect` with a generator. Similarly to `tmap`, specifying the output's element type is necessary to prevent performance losses.

```
x               = rand(1_000_000)
output          = similar(x)

foo(x)          = [log(a) for a in x]
foo_parallel1(x) = tcollect(log(a) for a in x)
foo_parallel2(x) = tcollect(eltype(x), log(a) for a in x)
```

```
julia> foo($x)
  3.231 ms (2 allocations: 7.629 MiB)
julia> foo_parallel1($x)
  1.489 ms (568 allocations: 16.958 MiB)
julia> foo_parallel2($x)
  336.948 μs (155 allocations: 7.642 MiB)
```

REDUCTIONS AND MAP-REDUCTIONS

`OhMyThreads` also offers multithreaded versions of `reduce` and `mapreduce`. They're respectively referred to as `treduce` and `tmapreduce`. These functions internally handle the race conditions inherent in reductions and address performance issues like false sharing. Notably, unlike `map`, these

functions can achieve optimal performance without requiring a specified output type.

```
x               = rand(1_000_000)

foo(x)          =  reduce(+, x)
foo_parallel(x) = treduce(+, x)
```

```
julia> foo($x)
  86.102 µs (0 allocations: 0 bytes)
julia> foo_parallel($x)
  29.542 µs (513 allocations: 43.047 KiB)
```

```
x               = rand(1_000_000)

foo(x)          =  mapreduce(log, +, x)
foo_parallel(x) = tmapreduce(log, +, x)
```

```
julia> foo($x)
  3.385 ms (0 allocations: 0 bytes)
julia> foo_parallel($x)
  389.624 µs (511 allocations: 43.000 KiB)
```

FOREACH AS A FASTER OPTION FOR MAPPINGS

The package also offers an implementation similar to for-loops through the function `tforeach`. Since we haven't covered the single-threaded version `foreach`, we begin by presenting it. The function follows a syntax identical to `map`, and is usually implemented using a do-block syntax, as shown below.

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> foo($x)
  3.329 ms (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    foreach(i -> output[i] = log(x[i]), eachindex(x))



    return output
end
```

```
julia> foo($x)
  3.251 ms (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    foreach(eachindex(x)) do i
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  3.265 ms (2 allocations: 7.629 MiB)
```

Despite the similarities of `tforeach` and `tmap`, `tforeach` is more performant. Furthermore, it doesn't incur a performance penalty when the output type isn't specified.

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> foo($x)
  3.281 ms (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    tmap(eachindex(x)) do i
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  1.868 ms (571 allocations: 24.589 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    tmap(eltype(x), eachindex(x)) do i
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  582.144 μs (158 allocations: 15.272 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    tmap(eltype(x), eachindex(x)) do i
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  582.144 μs (158 allocations: 15.272 MiB)
```

Just like `tmap`, `tforeach` offers the keyword arguments `nchunks` and `chunksize` to control the workload distribution among worker threads. For the illustration, we use a distribution analogous to the one used above for `tmap`.

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    tforeach(eachindex(x); nchunks = nthreads()) do i
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  340.708 μs (154 allocations: 7.642 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    tforeach(eachindex(x); chunksize = length(x) ÷ nthreads()) do i
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  358.567 μs (161 allocations: 7.643 MiB)
```

POLYESTER: PARALLELIZATION FOR SMALL OBJECTS

> **Warning!**
> All the code snippets below assume you executed `using Polyester` to load the package.

One key limitation of multithreading is its overhead due to the creation and scheduling of tasks. This issue can render parallelization impractical for smaller computational tasks, as the cost of thread management would outweigh any potential performance gain. Considering this, the application of multithreading is commonly reserved for objects sufficiently large to justify the cost.

The `Polyester` package addresses this limitation by implementing techniques that reduce the overhead. In this way, it becomes possible to parallelize objects that, otherwise, would be deemed too small to benefit from multithreading. Importantly, the package requires expressing the code to be parallelized as a for-loop.

To illustrate the benefits of the package, let's compare its performance to traditional methods. The following example considers a for-loop with 500 iterations, a relatively low number for applying multithreading. Indeed, the first tab shows that an approach based on `@threads` is slower than its single-threaded variant. In contrast, `Polyester` achieves comparable performance to the single-threaded variant, despite the low number of iterations. To use `Polyseter`, we simply need to prefix the for-loop with the `@batch` macro.

```
x = rand(500)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> foo($x)
  1.552 µs (1 allocations: 4.062 KiB)
```

```
x = rand(500)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> foo($x)
  9.362 µs (122 allocations: 16.672 KiB)
```

```
x = rand(500)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> foo($x)
  992.000 ns (1 allocations: 4.062 KiB)
```

For larger objects, it's worth noting that `Polyester` may not necessarily outperform (or underperform) alternative methods. In such cases, it's recommend benchmarking your particular application.

REDUCTIONS

`Polyester` also supports reduction operations. These can be implemented by prepending the for-loop with the expression `@batch reduce=(<tuple with operation and variable reduced>)`. Notably, Polyester's implementation has been designed to avoid common pitfalls of reductions, such as data races and false sharing, ensuring both correctness and performance. The following example illustrates its application.

```
x = rand(250)

function foo(x)
    output = 0.0

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

    return output
end
```

```
julia> @btime foo($x)
  745.289 ns (0 allocations: 0 bytes)
```

```
x = rand(250)

function foo(x)
    output = 0.0

    @batch reduction=( (+, output) ) for i in eachindex(x)
        output += log(x[i])
    end

    return output
end
```

```
julia> @btime foo($x)
  543.889 ns (0 allocations: 0 bytes)
```

We can also incorporate more than one reduction operation per iteration, as demonstrated below.

```
x = rand(250)

function foo(x)
    output1 = 1.0
    output2 = 0.0

    for i in eachindex(x)
        output1 *= log(x[i])
        output2 += exp(x[i])
    end

    return output1, output2
end
```

```
julia> @btime foo($x)
  1.241 µs (0 allocations: 0 bytes)
```

```
x = rand(250)

function foo(x)
    output1 = 1.0
    output2 = 0.0

    @batch reduction=( (*, output1), (+, output2) ) for i in eachindex(x)
        output1 *= log(x[i])
        output2 += exp(x[i])
    end

    return output1, output2
end
```

```
julia> @btime foo($x)
  630.302 ns (0 allocations: 0 bytes)
```

```
x = rand(250)

function foo(x)
    output1 = 1.0
    output2 = 0.0

    @batch reduction=( (*, output1), (+, output2) ) for i in eachindex(x)
        output1  = output1 * log(x[i])
        output2  = output2 + exp(x[i])
    end

    return output1, output2
end
```

```
julia> @btime foo($x)
  641.075 ns (0 allocations: 0 bytes)
```

LOCAL VARIABLES

`Polyester` also treats variables as local per iteration, unlike `@threads`.

```
function foo()
    out  = zeros(Int, 2)
    temp = 0

    for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo($x)
2-element Vector{Int64}:
 1
 2
```

```
function foo()
    out  = zeros(Int, 2)


    @threads for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo($x)
2-element Vector{Int64}:
 1
 2
```

```
function foo()
    out  = zeros(Int, 2)
    temp = 0

    @threads for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo($x)
2-element Vector{Int64}:
 2
 2
```

```
function foo()
    out  = zeros(Int, 2)
    temp = 0

    @batch for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo($x)
2-element Vector{Int64}:
 1
 2
```

SIMD + MULTITHREADING

> **Warning!**
> All the code snippets below assume you've already loaded the package with `using LoopVectorization`.

We've already covered the package `LoopVectorization` in the section about SIMD instructions. We now revisit this package to demonstrate its ability to combine SIMD with multithreading. The feature is achieved through integration with the `Polyester` package.

The primary approach to implementing the functionality involves the `@tturbo` macro, which provides a parallelized version of `@turbo`. Unlike the `@threads` macro, where the application of SIMD optimizations is left to the compiler's discretion, `@tturbo` automatically applies SIMD.

To illustrate the benefits of `@tturbo`, let's consider an example scenario where SIMD isn't applied automatically by `@threads`, despite that the operation is well-suited for this purpose.

```
x = BitVector(rand(Bool, 100_000))
y = rand(100_000)

function foo(x,y)
    output = similar(y)

    for i in eachindex(x)
        output[i] = ifelse(x[i], log(y[i]), y[i] * 2)
    end

    output
end
```

```
julia> foo($x)
  587.694 μs (2 allocations: 781.297 KiB)
```

```
x = BitVector(rand(Bool, 100_000))
y = rand(100_000)

function foo(x,y)
    output = similar(y)

    @threads for i in eachindex(x)
        output[i] = ifelse(x[i], log(y[i]), y[i] * 2)
    end

    output
end
```

```
julia> foo($x)
  80.625 μs (123 allocations: 793.906 KiB)
```

```
x = BitVector(rand(Bool, 100_000))
y = rand(100_000)

function foo(x,y)
    output = similar(y)

    @tturbo for i in eachindex(x)
        output[i] = ifelse(x[i], log(y[i]), y[i] * 2)
    end

    output
end
```

```
julia> foo($x)
  57.225 μs (2 allocations: 781.297 KiB)
```

The `@tturbo` macro is also available as a broadcasting version. Although the for-loop implementation could be more performant in some scenarios, the broadcasting variant significantly simplifies the syntax. Furthermore, it's particularly useful for parallelizing broadcasting operations,

since no built-in macro currently exists for this purpose. Below, we provide a simple example that demonstrates the improvement in readability achieved by using this variant.

```
x      = rand(1_000_000)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> foo($x)
  525.304 μs (2 allocations: 7.629 MiB)
```

```
x      = rand(1_000_000)

foo(x) = @tturbo log.(x) ./ x
```

```
julia> foo($x)
  524.273 μs (2 allocations: 7.629 MiB)
```

FLOOPS: PARALLEL FOR-LOOPS (*OPTIONAL*)

> **Warning!**
>
> All the code snippets below assume you've already loaded the package by executing `using FLoops`.

We conclude this section with a brief overview of the package `FLoops`. The presentation is labeled as optional since its use beyond simple applications could require some workarounds. Moreover, it appears not to be actively maintained.

The primary macro provided by the package is `@floop`, exclusively designed to parallelize for-loops. An example of its usage is provided below.

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> foo($x)
  3.353 ms (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

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

    return output
end
```

```
julia> foo($x)
  388.563 µs (157 allocations: 7.645 MiB)
```

`@floop` can also be used for reductions by including `@reduce` at the beginning of the line with a reduction operation. The macro addresses the inherent data race of reductions and avoids false sharing issues.

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

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

    return output
end
```

```
julia> foo($x)
  3.396 ms (0 allocations: 0 bytes)
```

```
x = rand(1_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> foo($x)
  1.314 ms (122 allocations: 13.234 KiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

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

    return output
end
```

```
julia> foo($x)
  370.835 μs (252 allocations: 17.516 KiB)
```

FOOTNOTES

[1] For example, `5 ÷ 3` would return `1`.