11g. Multithreading Packages

Martin Alfaro
PhD in Economics

INTRODUCTION

Parallelizing code may seem straightforward at first glance. However, once we start delving into its subtleties, it's rapidly revealed that an effective implementation can be a daunting task. 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 we've mastered the necessary skills for a correct implementation, manual parallelization often impairs the readability and maintainability of code.

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 performance issues like false sharing. We'll then examine the `Polyester` package. Thanks to its reduced overhead, this package excels at parallelizing workloads involving small objects. Finally, we revisit the `LoopVectorization` package to introduce the `@tturbo` macro, which combines the benefits of SIMD instructions with multithreading.

PACKAGE "OHMYTHREADS"

`OhMyThreads` is designed as a user-friendly package for seamlessly applying multithreading. Consistent with its minimalist approach, it introduces only a handful of essential functionalities that could easily belong to Julia's `Base`. Despite its simplicity, the package covers a wide range of operations, including reductions.

The package's primary goal is to make parallelization widely accessible, including users without deep expertise in multithreading. Indeed, its functions and macros automatically address common pitfalls, such as data races and performance issues like false sharing.

To achieve broad applicability, `OhMyThreads` extends familiar `Base` operations into the multithreaded domain through a set of higher-order functions. Each parallelized variant follows a simple naming convention: the original function name prefixed with `t`. For example, the parallel counterpart to `map`

becomes `tmap` in the package. At the same time, the package remains flexible by integrating with `ChunkSplitters`, allowing users to fine-tune how work is distributed across tasks. This customization is controlled through two keyword arguments: `nchunks` (or equivalently `ntasks`) to specify the number of subsets in the partition, and `chunksize` to define the number of elements in each partition.

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

PARALLEL MAPPING

`OhMyThreads` provides `tmap` as a multithreaded analogue of Julia's `map` function. The typical and most efficient way to call it is `tmap(foo, T, x)`, where `foo` is the function applied to each element of the collection `x`, and `T` denotes the element type of the resulting array. For instance, if the computation produces a `Vector{Float64}`, then `T` should be `Float64`.

Although you can omit the type parameter and write `tmap(foo, x)`, doing so introduces a performance penalty. This slowdown arises from a subtle source: the type instability of the underlying `Task` objects used to schedule work across threads. Because the compiler can't reliably infer the output type in this case, it's forced into less efficient code paths. To avoid this and recover full performance, you should explicitly specify the output element type. In practice, rather than hard-coding `T`, it's often clearer and safer to use `eltype(x)`, which ensures that the output array mirrors the element type of the input collection.

The package also offers an in-place variant `tmap!`, whose syntax is `tmap!(foo, output, x)`. Since the destination array is provided explicitly, the function already knows the output type, so there's no need to supply `T` to prevent performance issues.

The examples that follow illustrate how to use both `tmap` and `tmap!`. For context, we also include the corresponding results from `map` and `map!`, which serve 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)
```

`tmap` also allows us to control the work distribution among tasks through the keyword arguments `nchunks` and `chunksize`. These options are internally implemented via the package `ChunkSplitters`.

`tmap` also gives you fine-grained control over how work is divided among tasks through the keyword arguments `nchunks` and `chunksize`. These options rely internally on the `ChunkSplitters` package. 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 replicate the same behavior 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 passing anonymous functions into `tmap`, the do-block syntax comes in handy for keeping code readable. It enables the creation of multi-line functions, without the need to introduce a new function before applying `tmap`.

```
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 a parallel implementation of array comprehensions. Unlike the standard syntax of array comprehensions, the version from `OhMyThreads` combines a generator with a multithreaded variant of `collect` named `tcollect`.

As with `tmap`, specifying the output's element type is optional, abut necessary to avoid performance losses. The recommended syntax is therefore `tcollect(T, <generator>)`, where `T` denotes the element type of the output. A common practice is to use `eltype(x)` as `T`, with `x` being the variable iterated over in the generator. This ensures that the output type matches the input collection.

```
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` additionally provides multithreaded counterparts to `reduce` and `mapreduce`, respectively referred to as `treduce` and `tmapreduce`. These functions automatically handle the inherent race conditions arising in reductions. They also address performance issues such as false sharing. Unlike `tmap`, these functions are capable of achieving optimal performance without the need to explicitly specify an 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

`OhMyThreads` also offers a multithreaded version of `foreach` called `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.

```
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)
```

Similar to `tmap`, `tforeach` offers the keyword arguments `nchunks` and `chunksize` to control the workload distribution among worker threads. To illustrate, we use a work 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 the overhead introduced by the creation and scheduling of tasks. For smaller computational workloads, this overhead can outweigh any potential performance gain, rendering parallelization detrimental. As a result, multithreading is typically reserved for objects large enough to justify the cost.

The `Polyester` package addresses this limitation by providing a low-overhead implementation of for-loops. Its approach makes it possible to parallelize operations on objects that, otherwise, would be deemed too small to benefit from multithreading. To use `Polyester`, we simply prefix the for-loop with the `@batch` macro.

To illustrate its benefits, let's consider a for-loop with 500 iterations, a relatively low number for applying multithreading. The first tab below shows that an approach based on `@threads` is slower than its single-threaded variant. In contrast, `Polyester` achieves a comparable performance to the single-threaded variant, even with such a small workload.

```
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 workloads, it's worth noting that `Polyester` may not consistently outperform or underperform other multithreading approaches. Ultimately, performance will depend on the specifics of the computation and data involved. In such cases, the best practice is to benchmark your particular application.

REDUCTIONS

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

```
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)
```

It's possible to 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

Unlike macros like `@threads`, `Polyester` treats variables as local per iteration.

```
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 context of SIMD instructions. We now revisit this package to demonstrate its ability to combine SIMD with multithreading. The parallelization is achieved through its integration with `Polyester`.

The primary way to simultaneously implement SIMD and multithreading is via the `@tturbo` macro. This provides a parallelized version of `@turbo` in for-loops. Unlike the `@threads` macro, where the application of SIMD optimizations is left to the compiler's discretion, `@tturbo` enforces its application.

To illustrate the benefits of `@tturbo`, let's consider an example where SIMD isn't applied automatically by `@threads`, even when 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 also applies to broadcast operations. Offering this functionality is particularly valuable, as no built-in macro currently exists to parallelize broadcast expressions.

While applying `@tturbo` in a for-loop form often yields higher performance, the broadcast variant offers a much simpler and more concise syntax. The example below illustrates the improvement in readability stemming from this approach.

```
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 usage beyond simple applications may require some workarounds. In addition, the package doesn't appear to be actively maintained.

The primary macro offered by the package is `@floop`, exclusively designed for use with for-loops. An example of its application 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 placing `@reduce` at the beginning of the line containing the reduction operation. The macro addresses the inherent data race associated with and prevents false sharing.

```
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`.