Parallelism in Julia Spoke 18.S192
Homework 1

April 9, 2025

Please submit a PDF of your answers on GradeScope. This homework is graded. Use of ChatGPT or equivalent is allowed, perhaps encouraged (even for generating code, really!), but please write solutions to questions in your own words and comment on what LLM-generated code does and how you verified correctness, give credit, and please tell us, if you use an AI, which prompts for the AI worked and which ones did not work so well. (... and never never trust an AI without checking.)

You must submit all code written for this assignment.

You have two options for "multi-threading" for this homework, Julia's built-in `Threads.@threads` macro or the `OhMyThreads.jl` package: `https://juliafolds2.github.io/OhMyThreads.jl/dev/`. OhMyThreads can be easier to work with, especially if you want to adjust the number of tasks you are running at the same time. We recommend you read the docs for OhMyThreads if you choose to use it, but basic usage is below:

```
using OhMyThreads
OhMyThreads.@tasks for i in 1:5
    # body of your for loop
end

OhMyThreads.@task for j in rand(1000)
    OhMyThreads.@set chunksize = 100
    # body of your for loop
end

OhMyThreads.@tasks for k in eachindex(rand(10000))
    OhMyThreads.@set ntasks = 4 * Threads.nthreads()
    # body of your for loop
end
```

Due Monday, April 14, 11:59PM

1. **Do Operation Counts tell the whole story in the real world?:** In class, we looked at the speed of multiplying and adding two matrices of size $n \times n$.

   (a) What is the number of floating point ops (not in Big O-notation; include the first order constant) and number of data accesses for each operation?

   (b) Use `@belapsed` from Benchmarktools.jl to get the performance of both in function of the input matrix size. Compare the following four operations for small and large matrices, in powers of 2, from 2 to at least 2^{16} (and by preference much larger until you run out of patience or memory).

```
# define n =
eltype = Float32 #specify the data type (the default is Float64)
A = randn(eltype,n,n)
B = randn(eltype,n,n)
C = zeros(eltype,n,n) # (or C = similar(A)  also creates a )
mul!(C,A,B) #1. benchmark matmul for different n
C = A * B   #2. compare with the line above, what is the difference
            #in speed and why?
C .= A .+ B #3. benchmark matadd  for different n
C = A + B   #4. compare with the line above, what is the difference
            #in speed and why?
```

   Remember in Benchmarktools to use $ (Julia macro interpolation) for any data inputs.

   Please submit your code, and a table of computation times with n in the rows, and the four operations in the columns. Label your units (seconds or mseconds).

   (c) Make three plots using Plots.jl, use logarithmic scale where useful.

   - A plot of the absolute execution time for all four operations in function of input matrix size.
   - A plot of the execution rate for all four operations, i.e. the flops divided by the exectution time. (Express in readable units, e.g. GFlops where useful.)
   - A plot of the data access rate for all four operations, i.e. the number of data accesses divided by the execution time.

   Comment on what you see, specifically:

   - Compare operation 1 with 2 for matmul, and operation 3 with 4 for matadd. Which one is faster? Does the difference depend on matrix size and if yes, why?
   - Compare operation 1, matmul, with operation 3, matadd. Out of the three plots, which plot makes more sense for matmul and which one for matadd? What does this tell you about what is the bottleneck in performance for each operation?

(d) Implement Strassen's algorithm (many of you will have seen this, otherwise look online, or ask an AI for help) and compare times with `mul!`.

- Please submit your code, and a table of computation times with n in the rows, and the four operations in the columns. Label your units (seconds or mseconds).
- Submit a plot of the execution time of both in function of input size.
- What do you see compared to the previous algorithm? Can you think of an explanation(s) for the difference?

2. **Embarrassing Parallelism**

(a) Generate a `Vector` of random `Float64` using `v = rand(1000)`. Write a serial and threaded loop using `Threads.@threads` or `OhMyThreads.@tasks` to double each element (remember to put your code in a function! and verify the number of threads using `Threads.nthreads()`). Benchmark both using BenchmarkTools for very small matrix sizes (10 elements) up to extremely large matrices (at least approximately 1 minute computation time) in powers of 10 (or 100...).

- Please submit your code, and a table of computation times with n in the rows, and the two operations (threads / no threads) in the columns. Label your units (seconds or mseconds).
- Repeat the experiment for a different number of julia threads, e.g. 32, 8, 4 and 2. Submit a table of the computation times. As a reminder you can set the number of Julia threads at startup with `julia --threads=32` or set the environment variable `JULIA_NUM_THREADS`.
- Comment on your observations, in particular the difference between small and large inputs and the difference between different numbers of threads. Can you explain what you see? Is there anything surprising?

(b) Repeat the same experiment but fill your vector with square matrices and perform `A * A` on each matrix rather than doubling each element. You can use the `fill` function to create your vectors. Use `julia --threads=auto` for this experiment. Benchmark this operation with and without threads using BenchmarkTools for very small matrix sizes (4×4 elements) up to extremely large matrices in powers of 2 (or 4...).

- Please submit your code, and a table of computation times with n in the rows, and two operations (threads / no threads) in the columns. Label your units (seconds or mseconds).
- Submit a plot of the execution time of both functions in function of input size.
- What do you see on the plot: do the same conclusions as the previous experiment apply w.r.t. data size? What are other differences with the previous experiment? Can you think of an explanation(s) for the difference?

3. **Parallel Prefix**

   (a) Implement Algorithm 1 from the Wikipedia page on parallel prefix sum found here: `https://en.wikipedia.org/wiki/Prefix_sum`. Implement both a serial version and a version using `Threads.@threads` or `OhMyThreads.@tasks`. Submit your code.

   (b) Modify the algorithm to accept any element types or associative binary operators. Submit your code.

   (c) As you did in question 2 report on the performance of the algorithm for both small inputs and large input sizes. In addition compare the performance when the associative operator is relatively simple (like the `+` operator), to a relatively complex operator like matrix multiplication. Report a table with benchmarking results and a plot.

   (d) Report the multithreaded speed-up of parallelized element doubling (question 2a) in function of the input data sizes: report the ratio of serial vs threaded computation time in function of the input data size (rows). Report the same ratio for parallel prefix (question 3) in another column in the same table, and include a plot.

   (e) Is prefix sum embarrassingly parallel? Do we get a linear performance increase when we move from serial to parallel? Why (not)?