Parallelism in Julia Spoke 18.S192
Homework 1

April 8, 2025

Please submit a PDF of your answers on Canvas. 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.

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 rates of computation s 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!`. What do you see? 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` to double each element (remember to put your code in a function!). Compare and contrast the performance of both loops, Repeat your experiment for a very small vector and a much larger vector. Use BenchmarkTools in your comparisons.

   (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. Experiment with the size of the matrices and compare your results to the last experiment.

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`

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

   (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.

   (d) Is prefix sum embarrassingly parallel? Do we get a linear performance increase when we move from serial to parallel?