Special homework: Coding

This purpose of this homework is the following:

1. To encourage you to start coding in a performance-oriented language.
2. To get you started working on Stampede3 at TACC.
3. If you are already experienced with the topics presented during the early class periods, then this will give you something to do.

Grading: This is a special homework that will count for (few) extra points.

Tools: Use your knowledge, books, google, and AI coding assistance. Make use of any resource available, except your fellow students or other humans.

Terms: This homework uses terms not discussed in class, yet. Use google and AI agents to investigate.

Language: Use your favorite compiled language (C/C++, Fortran, etc.). If you do not 'speak' any compiled language, you may start with a different language of your choice.

Development platform: You can use your laptop, the VM, or Stampede3 for code development. Your final implementation must be tested on Stampede3. Timings must also be derived on Stampede3. We can discuss your code before you run the large examples on Stampede3, as you may run into problems.

Running experiments on Stampede3: Do not run the experiments on the login nodes. It is vital to use batch jobs on a shared resource like Stampede3.

General setup

1. Write code that implements the pseudo code shown below.
2. Run a few examples and report execution times.
3. Write clean and well documented code. Further below you will see several questions. Add the answers as comments to the code at the appropriate places.
4. Using a method in a class may incur overhead which may decrease performance. Think about an implementation that will limit or eliminate such overhead.
5. Add reasonable safeguards to prevent the code from failing.
6. Add timing for the experiments described below to your source code as comments.

Grading and procedure: Once you have written and completed the code, we meet in person during office hours (or by appointment) and discuss your work. This can be a multi-

step process, and we can meet a second time. The final submission window will close by mid-October. Keep in mind that I may not speak the language that you have chosen (very well). Be prepared to explain how your code works in detail.

Row-major v. column-major: The code you are writing deals with 2-dimensional arrays. This brings up the topic of row-major and column-major languages. As you will see below you will replace either a row or a column in a 2-d array. Pick the variant that you would expect to produce better performance.

Pseudo-code to be translated. This pseudo code does not use methods or even functions. The example shows plain code that you would find in a main program:

Select non-negative values for n, m, k.

All arrays elements are single precision floating point numbers.

Name the arrays x, b (for input), and y (for output).

```
n = 100; m = 50; k = 88   comment: k is either a column or a row
create x with (n by n) elements; preset all elements to 2
create b with (m) elements; preset to 1
create y with (n by n) elements; preset to 1
Loop over all array elements (n by n)
  y = (y + 2*x)/5
  If you are in row/column k then add b: y = y + b
        Note: This may apply only for the first (m) elements
              as the 1-d array (b) has (m) elements
End of loop
```

Instructions for your code regarding the implementation of classes and methods:

First class/object for the input arrays

```
x with (n by n) elements    (single precision)

b with (m)      elements    (single precision)

x = 2; b = 1

Add a method to create and initialize the input arrays
```

Second class/object for the output array

```
y with (n by n) elements    (single precision)

y = 1

Add a method to create and initialize the output array.
```

Add an additional method to perform the array manipulation

```
For the column or row (k) add the (m) elements of b to x to
calculate y (y = x + b). For all the other elements y = x.

In some mathematical notation:

Special elements:   y(k,…) = (y(k,…) + 2*x(k,…)) / 5 + b(…)
              or:   y(…,k) = (y(…,k) + 2*x(…,k)) / 5 + b(…)
All other elements: y(…,…) = (y(…,…) + 2*x(…,…)) / 5
              or:   y(…,…) = (y(…,…) + 2*x(…,…)) / 5
```

Your code:

1. Write code that implements the pseudo-code into the language of your choice.
    a. Use namespaces or modules as you see fit.
    b. Create a class/object for the input arrays. Add a method to initialize.
    c. Create a class/object for the output array. Add a method to initialize **y**, and a method that performs the calculation of array **y**.

2. Implementation details are left to you, but your code must meet these requirements:
    a. The code in green and red is to be placed in two classes, respectively.
    b. Design your implementation for maximum performance.
        i. Add either a partial column or row depending on the implantation language (row or column major).
        ii. Think about reducing the overhead that may come from invoking a method of a class (rather than having plain code in the main program shown in the pseudo-code).
        iii. Time the whole code by using `/usr/bin/time -p a.out`
        iv. Add output that shows how big the 3 arrays are. Choose appropriate units, KB, MB, or GB.
3. Answer these questions. Put the answers as comments in your code at the appropriate places:
    a. Where do you instantiate the objects? Why did you choose this particular location?
    b. Do you create your arrays on the stack or on the heap? Which 'place' would be more appropriate?
    c. How do you allocate the 2-dim arrays **x** and **y**? Why did you choose this particular method?
    d. Where would you expect your code to spend most of the time? Explain your reasoning.
    e. What safeguards did you add to prevent your code from crashing? Justify where you put the safeguards. Note that the input variables are non-negative and that you would not have to guard against those cases. But you should ensure that index `k` denotes a valid row/column in your (`n x n`) arrays.
4. Run these experiments on Stampede 3.
    a. `n=100  m=50  k=44`
    b. `n=1000  m=50  k=88`
    c. `n=25000  m=12345  k=12346`
    d. `n=90000  m=12345  k=12346`
5. Add output to your code to verify the correctness of the results.
    a. Calculate and print the sum of the elements of
        i. the first row
        ii. the k'th row
        iii. the first column
        iv. the k'th column

6. For small n only (n < 20), would you be able to create condensed output for array `y` that shows the values as a single-digit integer number? Output may look like this for a problem with `n=5`. This may help debugging problems.

```
11111
22211
11111
11111
11111
```

Final remarks:

1. If you are already an experienced programmer, then I hope you are having fun with this little exercise. I think that the implementation may take about 150 lines of code. Impress me with clean code, your implementation details, and how you extract the most performance.
2. If you are not so experienced with programming, then this exercise will undoubtedly be a bigger challenge.
3. Keep in mind that later in your career (if it has not happened already) you may be asked to implement some kind of algorithm that you are not familiar with. This homework will give you a taste of this.
4. I have little doubt that most of you will ask some AI tool to help you write this code, and this is perfectly fine for this exercise. This is a learning experience for all of us. I would like to gain some insight into how the AI was helpful to you, how you used it, and how long it took you to write the code. I would appreciate honest answers when we discuss this in person. The grade that you'll receive is based on the code itself, your ability to explain how the code works, and why you have chosen certain implementation details.