# ME 759
# High Performance Computing for Engineering Applications
# Assignment 2
# Due Thursday 02/06/2020 at 9:00 PM

Submit responses to all tasks which don't specify a file name to Canvas in a file called assignment2.txt, docx, pdf, rtf, odt (choose one of the formats). Also submit all plots on Canvas. Do not zip your Canvas submission. All *source files* should be submitted in the `HW02` subdirectory on the `master` branch of your `git` repo. The `HW02` subdirectory should have no subdirectories.

All commands or code must work on *Euler* with no modules loaded unless specified otherwise. Your commands and code may behave differently on your computer; please be sure to test on Euler before you submit.

Please submit clean code. Consider using a formatter like clang-format.
IMPORTANT: Before you begin, copy any provided files from `HW02` of the ME759-2020 repo.

---

1. a) Implement the `Scan` function in a file called `scan.cpp` with signature as in `scan.h`. You should write the exclusive scan[1] yourself and not use any library scan function.

   b) Write a file `task1.cpp` with a `main` function which (in this order)

   - Creates an array of `n` random `float`s however you like between -1.0 and 1.0. `n` should be read as the first command line argument as below.
   - Scans the array using your `Scan` function.
   - Prints out the time taken by your `Scan` function in *milliseconds*[2].
   - Prints the first element of the output scanned array.
   - Prints the last element of the scanned array.

   This file `task1.cpp` and its output won't be closely graded; simply make sure that you create reasonable input for the `Scan` function.

   - Compile: `g++ scan.cpp task1.cpp -Wall -O3 -o task1`
   - Run (where `n` is a positive integer): `./task1 n`
   - Example output (followed by newline):

   ```
   0.01
   0.0
   103.3
   ```

   c) On an Euler *compute node* (using a Slurm Job), run `task1` for each value `n` $= 2^{10}, x^{11}, \cdots, 2^{30}$ and generate a plot `task1.pdf` (with axis labels) which plots the time taken by your algorithm as a function of `n`. This is called a scaling analysis.

   Feel free to post your scaling analysis plot in case you want to help other colleagues get an idea of what they should obtain at the end of this exercise.

[1] Given an array $[a_0, a_1, \cdots, a_n]$, the exclusive scan function produces the array $[0, a_0, a_0+a_1, a_0+a_1+a_2, \cdots, a_0+a_1+\cdots+a_{n-1}]$. (In general, a scan can involve any other associative operation, not only $+$, like in this example.)

[2] Recall the document `timing.md`.

2. Convolutions[3] appear very prominently in image processing and in other fields like numerical solution of partial differential equations.

   a) Implement the `Convolve` function in a file called `convolution.cpp` with signature as in `convolution.h`.

$$g[x,y] = \sum_{i=0}^{m-1}\sum_{j=0}^{m-1} \omega[i,j] f[x+i-\frac{m-1}{2}, y+j-\frac{m-1}{2}] \qquad x,y = 0,\cdots,n-1$$

   where $f$ is the original image, $\omega$ is the mask, and $g$ is the result of the convolution. $m$ is the dimension of the square matrix $\omega$. There are several ways of dealing with edges, but here we'll just assume that $f[i,j] = 0$ whenever $0 \le i < n,\ 0 \le j < n$ is not satisfied.

   Example:

$$f = \begin{bmatrix} 1 & 3 & 4 & 8 \\ 6 & 5 & 2 & 4 \\ 3 & 4 & 6 & 8 \\ 1 & 4 & 5 & 2 \end{bmatrix}$$

$$\omega = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$g = \begin{bmatrix} 1 & 9 & 9 & 10 \\ 9 & 12 & 14 & 10 \\ 8 & 7 & 14 & 13 \\ 5 & 10 & 13 & 2 \end{bmatrix}$$

   b) Write a file `task2.cpp` with a `main` function which (in this order)

   - Creates an `n`×`n` `image` matrix (stored in 1D in row-major order) of random `float`s however you like. The value of `n` should be read as the first command line argument.
   - Creates a 3×3 `mask` matrix (stored in 1D in row-major order) of whatever mask values you like.
   - Applies the `mask` to `image` using your `Convolve` function.
   - Prints out the time taken by your `Convolve` function in *milliseconds*.
   - Prints the first element of the resulting convolved array.
   - Prints the last element of the resulting convolved array.

   The file `task2.cpp` and its output won't be closely graded; simply make sure that you create reasonable input for the `Convolve` function.

   - Compile: `g++ convolution.cpp task2.cpp -Wall -O3 -o task2`
   - Run (where `n` is a positive integer): `./task2 n`
   - Example output (followed by newline):

```
0.1
1.5
52.36
```

[3]See here for more on convolutions in image processing, just note that we use a slightly different formulation.

3. Implement in a file called `matmul.cpp` the four functions with signatures and descriptions as in `matmul.h` to produce the matrix product $C = AB$. For all of the cases, the array `C` that stores the matrix $C$ should be reported in row-major order.

   a) `mmul1` should have three `for` loops: the outer loop sweeps index `i` through the rows of `C`, the middle loop sweeps index `j` through the columns of `C`, and the innermost loop sweeps index `k` through the dot product of the `i`$^{th}$ row `A` with the `j`$^{th}$ column of `B`. Inside the innermost loop, you should have a single line of code which increments $\mathrm{C}_{ij}$. Assume that `A` and `B` are 1D arrays storing the matrices in row-major order.

   b) `mmul2` should also have three `for` loops, but the two outermost loops should be swapped relative to `mmul1`. That is the only difference between `mmul1` and `mmul2`.

   c) `mmul3` should have the `for` loops ordered as in `mmul1`. Assume that `A` is stored in row-major order and `B` is stored in column-major order. The only difference between this and `mmul1` should be the index calculations.

   d) `mmul4` should have the `for` loops ordered as in `mmul1`. Assume that `A` is stored in column-major order and `B` is stored in row-major order. The only difference between this and `mmul1` should be the index calculations.

   e) Write a program `task3.cpp` that accomplishes the following:

      - generates square matrices `A` and `B` of dimension at least 1000×1000 stored in row-major order.
      - computes the matrix product $C = AB$ using each of your functions (note that you may have to rearrange the input arrays to be in the correct storage pattern (row major or column major order) for each function in order to represent the same matrix). *Your result stored in matrix `C` should be the same no matter which function defined at a) through d) above you call.*
      - prints the number of rows of your input matrices.
      - for each `mmul` function in ascending order, prints the amount of time taken in *milliseconds* and the last element of the resulting `C`.
      - Compile command: `g++ task3.cpp matmul.cpp -Wall -O3 -o task3`
      - Run command: `./task3`
      - Sample expected output:

```
1024
1.23
2.365
1.23
2.365
1.23
2.365
1.23
2.365
```

   f) In a couple sentences, explain the difference that you see in the times for `mmul3` and `mmul4` *when running on an Euler compute node.* Make sure to discuss the hardware design and access patterns that cause this difference.