Subject: CAAM 420/520 – Homework 2
Date Posted: February 5, 2023
Date Due: 7:00pm, February 13, 2023 via Canvas

Total number of points available: (CAAM 420), (CAAM 520)

- **420:** *46 pts* Problems 1 and 2.
- **520:** *60 pts* Problems 1 and 3.

All problems in this homework refer to the PDE problem:

$$\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} = 0, \quad x > 0, y < 0$$

$$u(x=0,y) = \sin(2\pi y), \quad u(x, y=0) = \sin(2\pi x).$$

This problem is to be approximated on the domain $x \in [0,1], y \in [0,1]$ and discretized using the first order backwards finite difference formula:

$$\left.\frac{\partial f(x,y)}{\partial x}\right|_{x_i,y_j} \approx \frac{f(x_i,y_j) - f(x_{i-1},y_j)}{\Delta x}.$$

For a given $u_{i,j}$, this yields the equation:

$$u_{i,j} = u(x_i,y_j) = C_x u_{i-1,j} + C_y u_{i,j-1}$$

where:

$$C_x = \frac{\Delta y}{\Delta x + \Delta y},$$

$$C_y = \frac{\Delta x}{\Delta x + \Delta y},$$

and

$$\Delta x = \frac{1}{n_x}, \quad \Delta y = \frac{1}{ny}$$

Problem 1 : Domain Decomposition and Parallelizability *8 pts*

(a) *(3 pts)* Suppose you have four threads. Why would you not want to use the following domain decomposition to parallelize the FD problem?

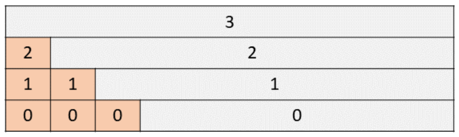

Figure 1. Example domain decomposition for Problem 1.a.

(b) *(5 pts)* Suppose T_s is the time it takes to synchronize and T_p is the time it takes to process a single FD node (i.e., the time to process a single $u_{i,j}$). If threads execute in a perfectly parallel fashion, then the time to process the fully-spun up region of the domain (assume the number of blocks in one dimension equals the number of threads) will be given by:

$$T_{full} = N_w(T_s + n_b T_p)$$

where N_w is the number of waves in the fully spun-up region and n_b is the number of elements per block. Notice that if we increase the number of blocks, n_b becomes smaller while N_w becomes larger.

When does synchronization contribute more time than computation to T_{full}? Smaller blocks allow us to have better parallelization by minimizing the spin-up and spin-down time. How does the cost of synchronization impact our choice of block sizes?

Problem 2 : 2D Wavefront Parallelization (420 only) *38 pts*

Associated Files: `main.cpp, wavefront.h`

Name your file: `<netID>_wavefront.cpp`

Expected compile command:

```
g++ -o hw2 -std=c++11 -fopenmp main.cpp <netID>_wavefront.cpp
```

Running the program:

```
export OMP_NUM_THREADS = <number of threads>
hw2 420 <Nx>
```

WARNING: do not modify `main.cpp` or `wavefront.h`. For testing you can write your own main file if you like and compile your program using the same command as above with your main file in place of `main.cpp`.

(a) *24 pts* Implement the wavefront parallelization in the function `wavefront420`where the number of blocks in the y-dimension, `Ny`, equals `num_threads`. Also implement the helper function `process_block`, which you must use in `wavefront420` to process each block. Note this is NOT the wrap-around algorithm discussed in class but the easier, "nice-case" domain decomposition. Make sure your implementation handles cases where the number of nodes may not be evenly divisible by the number of threads/number of blocks you choose.

The number of finite difference nodes, n_x and n_y, are given as constants in `wavefront.h`. Index `data` using the function `cartesian2flat`: $u_{i,j}$ = `data[cartesian2flat(i,j,ny)]`. Use the C math library, `cmath`, for sine.

(b) *7 pts* The code in main.cpp times your implementation. Report the (strong scaling) speed-up from using 1 to 8 threads and `Nx` = `Ny` = `num_threads`. Plot the results.

(c) *7 pts* Using 4 threads, time your code for `Nx = nx, nx/4, nx/16, nx/64, num_threads` and plot the results. Use log scale for the `Nx` axis.

Note: You can use MatLab, Excel, Julia, Python, etc for plotting. Your plots will be graded for presentation; make sure to label axes and give the plot a title. To get your code to compile, add an empty definition for the function `wavefront520` following to your .cpp file.

Problem 3 : 2D Wavefront Parallelization (520 only) *52 pts*

Associated Files: `main.cpp, wavefront.h`

Name your file: `<netID>_wavefront.cpp`

Expected compile command:

```
g++ -o hw2 -std=c++11 -fopenmp main.cpp <netID>_wavefront.cpp
```

Running the program:

```
export OMP_NUM_THREADS = <number of threads>
```

```
hw2 520 <Nx> <Ny>
```

WARNING: do not modify `main.cpp` or `wavefront.h`. For testing you can write your own main file if you like and compile your program using the same comman as above with your main file in place of `main.cpp`.

(a) *32 pts* Implement the wavefront parallelization in the function `wavefront520` with wrap around for when the number of blocks in each dimension does not equal `num_threads`. Also implement the helper function `process_block`, which you must use in `wavefront520` to process each block. Make sure your implementation handles cases where the number of nodes may not be evenly divisible by the number of threads/number of blocks you choose.

The number of finite difference nodes, n_x and n_y, are given as constants in `wavefront.h`. Index `data` using the function `cartesian2flat`: $u_{i,j} =$ `data[cartesian2flat(i,j,ny)]`. Use the C math library, `cmath`, for sine.

(b) *5 pts* In terms of n_x, n_y, N_x, N_y, and $N_T =$ `num_threads`, what fraction of the parallel program is spent in the spin-up and spin-down phases versus the fully-parallelized region? Assume every block takes the same amount of time to process and synchronize.

(c) *15 pts* The code in main.cpp times your implementation. Compute the (strong scaling) speed-up using 1 to 8 threads with `Nx = Ny = num_threads`. Repeat this with `Nx = Ny = 2*num_threads` and with `Nx = Ny = 3*num_threads`. Plot the three data sets on a single plot.

Note: You can use MatLab, Excel, Julia, Python, etc for plotting. Your plots will be graded for presentation; make sure to label axes and give the plot a title. To get your code to compile, add an empty definition for the function `wavefront420` following to your .cpp file.