Homework #4

ME 471/571

1. (**Winner, Winner, Chicken Dinner!**) Consider the following simplified version of the card game Blackjack. In this game, a dealer deals two cards to each player. Each player then places a bet, each player turns over their cards, and the winner of this single-round game of Blackjack is the player with the highest card total that doesn't exceed 21.

   In our implementation of this game, the dealer is Node 0. The dealer's job is to first create a shuffled deck of 52 cards, and then deal two cards to each processor (or "player"). Each player will then receive their cards, place their bets, and store the results of their hand to a common file. A Python script reads this common file, prints out the results and determines the winner.

   Your final output might look something like this:

   ```
   Winner, Winner, Chicken Dinner!

   Player  1 bet $ 74.50 on ( 6D,   KC) = 19 points
   Player  2 bet $ 50.00 on ( KD,   9C) = 22 points (bust!)
   Player  3 bet $ 26.00 on ( AS,   QH) = 13 points
   Player  4 bet $  1.50 on ( 8S,   8D) = 16 points
   Player  5 bet $ 77.00 on ( 8H,   6S) = 14 points

   Player  1 is a winner!  The winnings are $298.00.
   ```

   Here are some details.

   (a) A "card" consists is a number between 1 and 52.

   (b) When each player receives their two cards (i.e. two numbers in $[1, 52]$), they determine the suit (Spades(1-13), Hearts(14-26), Diamonds(27-39), Clubs(40-52)), and face value, using either the numeric value, or valuing the Ace as 1, the Jack as 11, the Queen as 12 and the King as 13.

   Each player then provides information to fill out the C-struct type

   ```
   typedef struct
   {
       char suits[2][10]  /* "spades","hearts","diamonds","clubs"           */
       char values[2][3]; /* "A", "2", "3", ... ,"10", "J", "Q", "K"        */
       int total;         /* Sum of two cards                               */
       float bet;         /* between $1.00 and $100.00, in inc. of $0.50$   */
   } struct_player_t;
   ```

   The players then collectively write out their player data to a common file (either as text or binary).

   (c) The bets that each player places can be taken as a dollar amount between \$1 and \$100, in increments of \$0.50.

   (d) The dealer is allowed to bet and play.

   (e) (**Optional.**) Allow the ace to be worth 1 or 11, depending on which helps the players hand.

   For this problem, you will create an MPI_Datatype derived for the player data represented in the C-struct above. To deal the cards, use the **MPI_Scatter** function. The scatter function is similar to broadcast, but instead of sending the same set of values to all processors, it distributes (or "deals") data to processors. Consult online MPI documentation to familiarize yourself with this command.

   To write out the data, create a one-dimensional array datatype using MPI_Type_create_subarray. You may write out your data as either text or binary. If you use binary, you can read the the data easily in Python by creating a Numpy "dtype" (see `numpy.dtype`). Keep in mind that even if you use binary, you still have to store the character values in `suits` and `values`.

2. (**Fractals.**) Design a fractal based on the CUDA code from the *Cuda by Example* text book. You will need to download this code from the Nvidia website (Google "Cuda by Example"). The codes used for this problem are from Chapter 4. Start with modified versions of the two fractal codes `julia_gpu.cu` and `julia_cpu.cu`, available on the course Github site. These modified codes replace the original visualization tools and instead create binary output files which you will load and visualize in the Jupyter notebook `plot_julia.png`. To compile these codes, the header files from the book will need to be available.

   For this problem, you will modify the CUDA code `julia_gpu.cu` to do the following.

   (a) The original code produces a fractal in $[-2, 2]$. Modify the code to create a fractal in a general square region $[x_{ll}, y_{ll}] \times [x_{ll} + d, y_{ll} + d]$, for a positive box height/width d.

   (b) The notebook `julia.ipynb` constructs a Julia set fractal using a coloring scheme based on the number of iterations it takes for the complex sequence of iterates to diverge. Implement this coloring scheme in the CUDA code `julia_gpu.cu`. Plot this in a notebook using the more advanced features of Matplotlib, demonstrated in `julia.ipynb`.

   (c) Create at least one Julia fractal using a different complex constant c.

   (d) Create a zoomed image of a fractal. For example, the image in `julia_gpu_zoom.png` is a zoom in the box of dimension 0.01×0.01, centered at $[1.156575, -0.1002331]$.

   (e) Compare timing results between the CPU and GPU codes. Use `nvprof` to profile the code :

   ```
   $nvprof julia_gpu
   ```

   This will report the timing for the kernel alone. Can you improve this time? Adjust the block dimensions to optimize the use of threads/warps in the kernel.

   Have fun!

3. (**Solving on nodes in 2d**) Does sharing nodes on processor boundaries extend to a node-centered iterative method in 2d? Show why or why not. Start with a simple test case of two 4×4 grids with a common boundary with four shared nodes. Assume Dirichlet boundary conditions. Extend what you find to 4 grids that share a corner. Then, show that this works in general for a domain partitioned into equally sized $N + 1 \times N + 1$ grids on nodes.

   For this problem, you do not need to write any code. Turn in either a hand written demonstration, or typeset your solution in Latex (preferred!).

4. (**CUDA**) One more problem involving choosing optimal CUDA execution configurations. Will not involve any coding, but will involve running code from class.