1. I chose to do the 1D partition. I ended up creating horizontal slices of A and vertical slices of B. It was the easiest to have work being done in every process at the same time, unlike blocking which would not always have working being done on every lock step.
2. I wrote a simple Python3 script. It writes a file that outputs random numbers generated from a to b inclusively. The file’s numbers are separated by a space. Every time there is a new row, a new line character is appended to the string. See example below:

0 8 7

7 2 9

6 3 4

1. I simply modified the supplied matrix sequential code from class to accept file input in my format listed about. I verified the answer with an online tool as well. Filename: matrix-seq.c
2. See matrix.c
3. See Analytics excel sheet
4. 1. On the ginormous matrix, we saw a really large improvement with each additional processor. As for the large & modest matrix, the matrices were not large enough to have a real impact due to all the code performing send/receives with MPI.
   2. No surprises
   3. Doing the matrix multiply with MPI. I think had I implemented another MPI problem without matrix multiplication, I would have figured it out much faster because learning MPI took quite a bit of time. It’s not very plug and play.