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. I tried to complete it, but never got the code working. It performs an abort trap somewhere after do the multiplication on each slice. However, in the attached matrix.c, you can see my code.
3. See Analytics excel sheet
4. 1. Performance increased from 1 to 2 processors, but decreased with 4 processors. My belief is that this has to do with how Mac OS was scheduling since I believe that I have 4 physical cores. As the matrix size increased, the performance increased. Having an additional 2 processors for the large/ginormous matrix really seemed to improve things.
   2. That the time decreased with 4 processors, I really thought it would be a logarithmic decrease.
   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.