Phase 1: Setting up the necessary variables and data structures
1. Initialized an input array for each process and then stored random numbers inside these arrays. Now the arrays contain unsorted random numbers of size (total num / total # process) 2. Similarly initialized receive buffer for each process of size equal to the total numbers to be sorted (since a receive buffer of a process can receive all the numbers from a process: if the splitters is not doing its job) 3. Then initialized an array to store the sample. The size of each sample array is 1% of the total numbers to be sorted. Also, the sample array contains element from the starting position of the corresponding input array of the process (Since the numbers are randomly distributed in the unsorted input array: choosing any starting point of the sample array does not matter) 4. Initialized the splitter arrays for each process with size equal to the size of the process.

Phase 2: Sorting the Sample Using Odd-Even Transposition
1. I basically copy pasted the algorithms that I used in assignment 7 to sort the sample array of each process using odd-even transposition.

Phase 3: Calculating the splitters
1. As the sample arrays are now sorted, stored the minimum value and the maximum value in variables my_min and my_max respectively 2. All the process sent their max to the process on the right and received the min value of the process to their left. Except, process 0 only sent its max value to process 1 and did not receive any value from any process. Also, process of the highest rank only received max value of the process on its left and did not send its max value to any other process. After receiving the necessary variables, the process (except process 0) calculated the corresponding splitters. 3. After every process had calculated their respective splitters, MPI AllGather was used to distribute the final splitter array to every process. Afterwards, the size of the splitter array was increased by 1 and 0 was added at the end of the array.

Phase 4: Sample Sort

1. The code basically follows the implementation of the pseudocode of the sample sort of second implementation (described in the slides of April 14)
2. Sorted each input array (initialized on step 1 of phase 1) using the built in sort feature of NumPy array (np.sort(my_input_array))
3. Initialized bitmask to half of the total number of processes and same for the variable which_splitters. Ran a while loop (bitmask >=1)
4. Each process obtained the rank of its partner by xoring with the current value of bitmask
5. Inside of the while loop number of functions were involved. Get_send_args calculated the number of elements each process must send to its partner. Using the count, each process also calculates the offset (starting point of sending the elements) by using the simple formula: local_n – count (if my_rank < partner) and offset equal to 0 when partner < my_rank, where local_n is the size of the input array before sending the elements to its partner
6. Cut bitmask to half and increase the value of which_splitter by bitmask for process whose rank is > its partner and decrease the value of which_splitter by bitmask if their rank is less than their partner
7. After the necessary information is calculated, each process sends and receive the necessary elements from its partner based on the current value of which_splitter (which corresponds to the element position in the splitter array)
8. Then MPI.Get_count is used to figure out how many elements a process has received from its partner. The receiving count is necessary for the Merge(…) function as it exactly knows how many elements to consider in the receive arrays (as the size of the receiving buffer is larger than the receiving count and Merge function will not know when to stop looking for elements in the receiving buffer)
9. The Merge functions basically concatenates (and sort) the receive array and the input array and return the resulting array. The offset of the input array and the updated size

(after sending elements to its partner) and the receive count is used to execute the above step

10. The return value of the Merge function is taken by the input array.
11. Now the input array contains the elements after 1 iteration of the while loop
12. The iteration continues until bitmask < 1
13. The total run time is calculated and is used to derive conclusions by running the program with different number of processes.
14. Afterwards Gather is used to create recv_count_array to store the number of elements of all process that will be sent to process 0. Similarly, the necessary offset array is built for process 0 by using the values in the recv_count_array
15. Lastly, Gatherv is used to gather all the sorted elements to process 0. At the end, the resulting array contains the sorted elements.

Results of running the program

With 2 processes: Average = 0.03s

```
C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 2 python p.py
Total Runtime is: 0.028698800000711344

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 2 python p.py
Total Runtime is: 0.050877700006822124

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 2 python p.py
Total Runtime is: 0.03897399999550544

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 2 python p.py
Total Runtime is: 0.03283499999088235

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 2 python p.py
Total Runtime is: 0.02034329999878537
```

With 4 processes: Average = 0.05s

```
C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 4 python p.py
Total Runtime is: 0.06102310000278521

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 4 python p.py
Total Runtime is: 0.044515299989143386

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 4 python p.py
Total Runtime is: 0.05113430001074448

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 4 python p.py
Total Runtime is: 0.054383900001994334

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 4 python p.py
Total Runtime is: 0.06580319999193307
```

With 8 processes: Average = 0.03s

```
C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 8 python p.py
Total Runtime is: 0.028430299993488006

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 8 python p.py
Total Runtime is: 0.03482459999213461

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 8 python p.py
Total Runtime is: 0.05267009999079164

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 8 python p.py
Total Runtime is: 0.05609399999957532

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 8 python p.py
Total Runtime is: 0.043254699994577095
```

With 16 processes: Average = 0.10s

```
C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 16 python p.py
Total Runtime is: 0.12249069999961648

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 16 python p.py
Total Runtime is: 0.0895814999967115

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 16 python p.py
Total Runtime is: 0.17330479998781811

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 16 python p.py
Total Runtime is: 0.07920389999344479

C:\Users\zeeha\Desktop\Documents\Courses\CSCI381-D\Project>mpiexec -n 16 python p.py
Total Runtime is: 0.08220390000496991
```

Conclusion: setting p to greater than or equal to 16 is likely to deteriorate the runtime due to communication overheads. However, the system has 8 logical processors, so the long run time could because of mpi trying to run 16 processes while there are only 8 logical processors.
As we can see the lowest time is scored for p = 2 or 8. Most probably, the ideal number of cores for this problem lies between 2 and 8.