1 Question 1

1.1 Parallel MPI version

I designed the code to divide the search space based on the number of processes. Each process starts at its rank and moves in steps equal to the total number of processes. This ensures that the search space is evenly distributed on all processes, avoiding overlaps. If a process finds the password, it sets a flag to 1 and records its rank. All processes check whether the password has been found. If it has, they identify the process with the lowest rank that found the password and broadcast it from that rank. Table 1 presents the results.

Password	**#Processes**	**Time (seconds)**
WLU	1	34.19
	5	7.92
	10	4.35
cp6	1	8.55
	5	3.03
	10	1.71
123	1	6.82
	5	1.29
	10	0.73
XYZ	1	39.41
	5	8.92
	10	5.05

Table 1: Comparison of Time Using Different Passwords and Varying Numbers of Processes

Using the parallel method, as the number of processes increases, the time required to find the password decreases. The program scales with the number of processes because each process handles a smaller portion of the password search space. Once the password is found, all processes are informed and terminate, preventing wasted computation. However, it's important to note that the program's performance depends on the number of available processors and the size of the password search space. For small search spaces, the

communication overhead of MPI may outweigh the benefits, but for larger search spaces, the parallel approach should result in significant speedups. Additionally, the password's location within the search space affects the search time. For example, a password like '123' located near the beginning of the search space will be found faster since fewer combinations need to be evaluated. On the other hand, passwords near the end, such as 'WLU', require more iterations and lead to longer search times.

1.2 4 character password

The results of running the parallel program with a 4-character password are illustrated in Table 2, which shows that as the number of processes increases, the time taken decreases. However, it takes more time than the 3-character password due to the greater number of possible combinations. These results are from server 1.

Password	**#Processes**	**Time (seconds)**
L4uD	1	1002.04
	5	251.40
	10	138.101

Table 2: Comparison of Time Using Varying Numbers of Processes

1.2.1 Server 2

Using course server 2, these are the results:

- **Job Number**: 2246.
- **Directory**: /home/sahmed147/Assignment1/submit_job.sh.
- **Job Runtime**: 86.311686 seconds, approximately 1 minute and 27 seconds.
- **Password Found**: L4uD.

2 Question 2

For question two, the table 3 summarizes the results after running the program with matrix sizes of up to 1000 using different numbers of processes.

Matrix Size	#Processes	Time (seconds)
1000	100	1.73
	50	0.7425
	25	0.43
8	8	0.0017
	4	0.00045
	2	0.00026

Table 3: Results after running the program with different matrix sizes and different numbers of processes.

The results show that when we decrease the number of processes, the runtime decreases, possibly due to communication overhead between processes, such as sending/receiving row and column data with process 0 using `MPI_Send` and `MPI_Recv`. As the number of processes increases, the communication overhead may outweigh the benefits.

Steps to run the code:

1. First, run the matrix generator:

   ```
   ./matrix_generator n
   ```

   where `n` is the matrix size.

2. Then, run the program read_write_matrix:

   ```
   mpirun -np p ./read_write_matrix n
   ```

   where `p` is the number of processes, and `n` is the matrix size.