Week-1

1. mpicc mpi01.c -o mpi01: Compiling the mpi file

```
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$ mpicc mpi01.c -o mpi01
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$ ls
mpi01  mpi01.c  mpi02.c  mpi03.c
```

mpiexec ./mpi01: Running the compiled mpi executable

```
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$ mpiexec ./mpi01
I am 0 of 1
```

2. mpicc mpi02.c -o mpi02: Compiling

```
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$ mpicc mpi02.c -o mpi02
```

mpiexec -n 2 -oversubscribe ./mpi02: Executing with 2 processes

```
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$ mpiexec -n 2 -oversubscribe ./mpi02
This program needs to run on exactly 3 processes
```

The program does not run as there are two ranks 0 and 1 for the two processes. And in our code, we have defined rank 1 and 2. Rank 2 does not exist so the error says program needs to run on exactly 3 threads.

mpiexec -n 3 -oversubscribe ./mpi02

```
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$ mpiexec -n 3 -oversubscribe ./mpi02
Process 1 received 9
Process 2 received 17
```

In this case, there are 3 processes. So there are ranks 0, 1 and 2. In our code, we have ranks up-to 2. So the program runs sucessfully.

3. The part of the code that defines the condition that exactly 3 processes should be used has been removed.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
  int size, rank;

  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank ==0){
      int x = 9;
      int y = 17;
      for(int i=1; i<size; i++){
        MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Send(&y, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
      }
    } else {
      int number;
      MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Process %d received %d\n", rank, number);
    }
  }
  MPI_Finalize();

  return 0;
}
```

```
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$ mpicc mpi02.c -o mpi02
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$ mpiexec -n 2 -oversubscribe ./mpi02
Process 1 received 9
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12149] *** An error occurred in MPI_Send
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12149] *** reported by process [1857028097,0]
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12149] *** on communicator MPI_COMM_WORLD
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12149] *** MPI_ERR_RANK: invalid rank
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12149] *** MPI_ERRORS_ARE_FATAL (processes in this communicator will now abort,
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12149] ***    and potentially your MPI job)
```

An error is seen saying invalid rank when we run it with two processes. As mentioned earlier, the program does not run as there are two ranks 0 and 1 for the two processes. And in our code, we have defined rank 1 and 2. Rank 2 does not exist so the error says invalid rank.

```
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$ mpiexec -n 3 -oversubscribe ./mpi02
Process 2 received 17
Process 1 received 9
```

The program runs successfully with 3 processes as there are ranks 0, 1 and 2. In our code, we have ranks up-to 2. So the program runs successfully.

```
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$ mpiexec -n 4 -oversubscribe ./mpi02
Process 1 received 9
Process 2 received 17
^C
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] *** Process received signal ***
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] Signal: Segmentation fault (11)
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] Signal code: Address not mapped (1)
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] Failing at address: (nil)
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] [ 0] /lib/x86_64-linux-gnu/libc.so.6(+0x45250) [0x7edb80645250]
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] [ 1] /lib/x86_64-linux-gnu/libpmix.so.2(PMIx_server_finalize+0xa89) [0x7edb72875779]
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] [ 2] /usr/lib/x86_64-linux-gnu/openmpi/lib/openmpi3/mca_pmix_ext3x.so(ext3x_server_finalize+0x3b6) [0x7edb7ffadfb6]
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] [ 3] /lib/x86_64-linux-gnu/libopen-rte.so.40(pmix_server_finalize+0xb2) [0x7edb80a30272]
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] [ 4] /usr/lib/x86_64-linux-gnu/openmpi/lib/openmpi3/mca_ess_hnp.so(+0x6776) [0x7edb804b5776]
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] [ 5] /lib/x86_64-linux-gnu/libopen-rte.so.40(orte_finalize+0x64) [0x7edb80a04a44]
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] [ 6] mpiexec(+0x134f) [0x5f0f02e0834f]
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] [ 7] /lib/x86_64-linux-gnu/libc.so.6(+0x2a3b8) [0x7edb8062a3b8]
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] [ 8] /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0x8b) [0x7edb8062a47b]
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] [ 9] mpiexec(+0x1415) [0x5f0f02e08415]
[prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:12254] *** End of error message ***
Segmentation fault (core dumped)
```

When 4 processes are defined, there are ranks 0, 1, 2 and 3. But in our code, the defined ranks are only up-to 2. The rank 0 sends the information to rank 1 and 2 but it does not send anything to rank 3. The rank 3 never receives the message so the program keeps executing until rank 3 receives any information.

4. To solve the above issue, we can modify the code such that:

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
  int size, rank;

  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if(rank ==0){
      int x = 9;
      int y = 17;
        for(int i=1; i<size; i++){
        MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Send(&y, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
        }
    } else {
      int number;
      MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Process %d received %d\n", rank, number);
    }
  MPI_Finalize();

  return 0;
}
```

Now as we can see, when the program is run with 4 processes, the program runs and terminates successfully. The loop ran until the size, so the rank mismatch problem is solved.

```
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$ mpiexec -n 4 -oversubscribe ./mpi02
Process 2 received 9
Process 3 received 9
Process 1 received 9
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$
```

5.

```
prashun@prashun-ASUS-TUF-Gaming-A15-FA506IC-FA506IC:~/week-1/tutorial$ mpiexec -n 3 -oversubscribe ./mpi03
Received 11 from process 1
Received 12 from process 2
```

Here, the program executes only after a certain time as there is a sleep function for rank 1. So other ranks won't be able to send/receive messages until rank 1 sends a message.

```c
#include <stdio.h>
#include <mpi.h>
#include <unistd.h>

int main(int argc, char** argv) {
  int size, rank;

  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if(size != 3) {
    if(rank == 0) {
      printf("This program needs to run on exactly 3 processes\n");
    }
  } else {
    if(rank ==0){
      int x, y;
      MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Received %d from process %d\n", x, 1);
      MPI_Recv(&y, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Received %d from process %d\n", y, 2);
    } else {
      if(rank == 1){
        usleep(5000000);
      }
      int number = rank + 10;
      MPI_Send(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
  }
  MPI_Finalize();

  return 0;
}
```

That can be solved using MPI_ANY_SOURCE which makes the program run asap as it receives whichever message it can receive.

6.

```c
#include <stdio.h>
#include <mpi.h>

int is_prime(int num) {
    if (num < 2){
        return 0;
    }
    for (int i = 2; i * i <= num; i++) {
        if (num % i == 0){
                return 0;
        }
    }
    return 1;
}
```

```c
int main(int argc, char **argv) {
    int rank, size;
    int nstart = 1, nfinish = 100;
    int chunk_size, start, end;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    chunk_size = (nfinish - nstart + 1) / size;

    start = nstart + rank * chunk_size;
    if (rank == size - 1) {
    end = nfinish;
    } else {
        end = start + chunk_size - 1;
    }

    printf("Process %d checking numbers from %d to %d\n", rank, start, end);

    for (int i = start; i <= end; i++) {
        if (is_prime(i)) {
            printf("Process %d: %d\n", rank, i);
        }
    }

    MPI_Finalize();
    return 0;
}
```

This program calculates prime numbers between two values, `nstart` and `nfinish`, using MPI to parallelize the task across multiple processes. Initially, it divides the range into chunks, with each process assigned to check a specific subset of numbers for primality. The range is split based on the number of processes, ensuring each process handles an equal share of the work. The program

checks if each number in a process's assigned range is prime by testing divisibility up to the square root of the number, which improves efficiency. If a process identifies a prime, it prints it along with its process ID. The last process handles any remaining numbers if the range doesn’t divide evenly. After completing the task, all processes finalize, and the program ends. This parallel approach speeds up the computation by distributing the work across multiple CPU cores, making it much faster than running the task with a single process.