GENERAL EXECUTION STEPS:-

LOGGING INTO PUTTY:-

1) Open PuTTY
2) Enter the IP "**172.16.13.15**" [IP may/may not be changed during exam]
3) Once you are inside the console, type the following credentials
   Username - cse7<your_section(in lowercase)><your_roll_number(01-61)>
   Password - sastra123
   [Note: Credentials may be changed during the exam]

MPI PROGRAM CREATION, COMPILATION, AND EXECUTION:-

1) To create a program, type the following command
   vi <filename>.cu
2) Once you are inside the file, press I to go to INSERT MODE(edit mode)
3) After tying the code, press the following combo sequentially to save the file
   Esc(escape) + : (colon) -> then in prompt type wq
4) To compile a CUDA program, type the following command
   mpicc <filename>.cu
5) To run the file, use the following command,
   mpirun -np <number of processors to be used> a.out

EXP 7 - CHAT SERVER APPLICATION

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

#define MAX_MESSAGE_LENGTH 1024

int main(int argc, char** argv) {
   int num_processes, process_rank;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
   MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);

   if (process_rank == 0) {
      char message[MAX_MESSAGE_LENGTH];
      printf("\nEnter a message: ");
      fflush(stdout);  // Flush the output buffer to display the prompt immediately
      fgets(message, MAX_MESSAGE_LENGTH, stdin);
      MPI_Bcast(message, MAX_MESSAGE_LENGTH, MPI_CHAR, 0,
MPI_COMM_WORLD);
   } else {
      char message[MAX_MESSAGE_LENGTH];
      MPI_Bcast(message, MAX_MESSAGE_LENGTH, MPI_CHAR, 0,
MPI_COMM_WORLD);
      printf("\nReceived message from root process: %s", message);
   }

   int i;

   for (i = 0; i < num_processes; i++) {
      if (process_rank != i) {
         char send_message[MAX_MESSAGE_LENGTH];
         sprintf(send_message, "Hello from process %d!", process_rank);
         MPI_Send(send_message, MAX_MESSAGE_LENGTH, MPI_CHAR, i, 0,
MPI_COMM_WORLD);
```

```
            char recv_message[MAX_MESSAGE_LENGTH];
            MPI_Recv(recv_message, MAX_MESSAGE_LENGTH, MPI_CHAR, i, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("\nReceived message from process %d: %s\n", i, recv_message);
        }
    }

    MPI_Finalize();
    return 0;
}
```

```
[cse7b27@raman-pac MPI Programs]$ mpicc ex7.c
[cse7b27@raman-pac MPI Programs]$ mpirun -np 4 a.out

Enter a message: hello

Received message from process 1: Hello from process 1!

Received message from process 2: Hello from process 2!

Received message from process 3: Hello from process 3!

Received message from root process: hello

Received message from process 0: Hello from process 0!

Received message from process 2: Hello from process 2!

Received message from process 3: Hello from process 3!

Received message from root process: hello

Received message from process 0: Hello from process 0!

Received message from process 1: Hello from process 1!

Received message from process 3: Hello from process 3!

Received message from root process: hello

Received message from process 0: Hello from process 0!

Received message from process 1: Hello from process 1!

Received message from process 2: Hello from process 2!
[cse7b27@raman-pac MPI Programs]$
```

EXP 8 - MUTUAL EXCLUSION

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

```
#include <unistd.h>  // For sleep function

#define REQUEST 1
#define REPLY 2
#define RELEASE 3
#define MAX_PROCESSES 10

int timestamp = 0;
int num_processes, process_rank;
bool requesting = false;
bool in_critical_section = false;
int replies_count = 0;
bool deferred_reply[MAX_PROCESSES];

void send_message(int dest, int tag) {
    if (dest >= 0 && dest < num_processes) {
        MPI_Send(&timestamp, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
    }
}

void receive_message(int* recv_timestamp, int* source, int* tag) {
    MPI_Status status;
    MPI_Recv(recv_timestamp, 1, MPI_INT, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    *source = status.MPI_SOURCE;
    *tag = status.MPI_TAG;
}

void handle_request(int source, int recv_timestamp) {
    bool grant_permission = false;

    if (!in_critical_section && !requesting) {
        grant_permission = true;
    } else if (recv_timestamp < timestamp || (recv_timestamp == timestamp &&
source < process_rank)) {
        grant_permission = true;
    }
```

```
    if (grant_permission) {
        send_message(source, REPLY);
    } else {
        deferred_reply[source] = true;
    }
}

void handle_reply() {
    replies_count++;
    if (replies_count == num_processes - 1) {
        in_critical_section = true;
        printf("Process %d in critical section\n", process_rank);
        sleep(1);  // Simulate time spent in the critical section
    }
}

void handle_release(int source) {
    printf("Process %d received RELEASE from process %d\n", process_rank,
source);
}

void request_critical_section() {
    requesting = true;
    timestamp++;
    replies_count = 0;

    int i;

    for (i = 0; i < num_processes; i++) {
        if (i != process_rank) {
            send_message(i, REQUEST);
        }
    }
}

void release_critical_section() {
```

```
    printf("Process %d releasing critical section\n", process_rank);
    in_critical_section = false;
    requesting = false;

    int i;

    for (i = 0; i < num_processes; i++) {
        if (deferred_reply[i]) {
            send_message(i, REPLY);
            deferred_reply[i] = false;
        }
    }


    for (i = 0; i < num_processes; i++) {
        if (i != process_rank) {
            send_message(i, RELEASE);
        }
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);

    int i;

    for (i = 0; i < MAX_PROCESSES; i++) {
        deferred_reply[i] = false;
    }

    if (process_rank == 0) {
        sleep(1);  // Process 0 starts first
    }
    request_critical_section();
```

```
    while (true) {
        int recv_timestamp, tag, source;
        receive_message(&recv_timestamp, &source, &tag);

        timestamp = (timestamp > recv_timestamp) ? timestamp + 1 :
recv_timestamp + 1;

        switch (tag) {
            case REQUEST:
                printf("Process %d received REQUEST from process %d\n",
process_rank, source);
                handle_request(source, recv_timestamp);
                break;
            case REPLY:
                printf("Process %d received OK from process %d\n", process_rank,
source);
                handle_reply();
                break;
            case RELEASE:
                handle_release(source);
                break;
        }

        if (in_critical_section) {
            release_critical_section();
            break;
        }
    }

    MPI_Finalize();
    return 0;
}
```

```
[cse7b27@raman-pac MPI Programs]$ mpicc ex8.c
[cse7b27@raman-pac MPI Programs]$ mpirun -np 4 a.out
Process 1 received REQUEST from process 3
Process 1 received OK from process 3
Process 3 received REQUEST from process 1
Process 3 received OK from process 1
Process 2 received REQUEST from process 1
Process 2 received OK from process 1
Process 2 received REQUEST from process 3
Process 2 received OK from process 3
Process 1 received REQUEST from process 2
Process 1 received OK from process 2
Process 3 received REQUEST from process 2
Process 3 received OK from process 2
Process 2 received REQUEST from process 0
Process 2 received OK from process 0
Process 2 in critical section
Process 3 received REQUEST from process 0
Process 3 received OK from process 0
Process 3 in critical section
Process 0 received REQUEST from process 1
Process 0 received REQUEST from process 2
Process 0 received REQUEST from process 3
Process 0 received OK from process 2
Process 0 received OK from process 3
Process 1 received REQUEST from process 0
Process 1 received OK from process 0
Process 1 in critical section
Process 0 received OK from process 1
Process 0 in critical section
Process 2 releasing critical section
Process 3 releasing critical section
Process 1 releasing critical section
Process 0 releasing critical section
[cse7b27@raman-pac MPI Programs]$
```

EXP 9 - GROUP COMMUNICATION

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

```
int main(int argc, char** argv) {
    int num_processes, process_rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);

    int array_size = num_processes;  // Set array size equal to the number of
processes
    char message[100];

    if (process_rank == 0) {
        printf("Enter a message to broadcast: ");
        fflush(stdout);  // Force the output to be displayed immediately
        fgets(message, 100, stdin);

        printf("Array size set to number of processes: %d\n", array_size);
        printf("Enter %d elements for the array:\n", array_size);
        fflush(stdout);  // Force the output to be displayed immediately
    }

    MPI_Bcast(&array_size, 1, MPI_INT, 0, MPI_COMM_WORLD);

    int numbers[array_size];  // Declare the numbers array with size equal to
number of processes

    if (process_rank == 0) {
        int i;
        for (i = 0; i < array_size; i++) {
            scanf("%d", &numbers[i]);
        }
    }

    MPI_Bcast(message, 100, MPI_CHAR, 0, MPI_COMM_WORLD);
    printf("Process %d received message: %s\n", process_rank, message);
```

```
    int recv_number;

    MPI_Scatter(numbers, 1, MPI_INT, &recv_number, 1, MPI_INT, 0,
MPI_COMM_WORLD);

    int result = recv_number * recv_number;
    printf("Process %d received %d and computed its square: %d\n",
process_rank, recv_number, result);

    int gathered_results[array_size];

    MPI_Gather(&result, 1, MPI_INT, gathered_results, 1, MPI_INT, 0,
MPI_COMM_WORLD);

    if (process_rank == 0) {
        printf("Gathered results: ");
        int i;
        for (i = 0; i < array_size; i++) {
            printf("%d ", gathered_results[i]);
        }
        printf("\n");
    }

    int sum_of_squares;
    MPI_Reduce(&result, &sum_of_squares, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    if (process_rank == 0) {
        printf("Sum of squares: %d\n", sum_of_squares);
    }

    MPI_Finalize();
    return 0;
}
```

```
[cse7b27@raman-pac MPI Programs]$ mpicc ex9.c
[cse7b27@raman-pac MPI Programs]$ mpirun -np 4 a.out
Enter a message to broadcast: hello
Array size set to number of processes: 4
Enter 4 elements for the array:
1 2 3 4
Process 0 received message: hello

Process 0 received 1 and computed its square: 1
Process 1 received message: hello

Process 1 received 2 and computed its square: 4
Process 2 received message: hello

Process 2 received 3 and computed its square: 9
Gathered results: 1 4 9 16
Sum of squares: 30
Process 3 received message: hello

Process 3 received 4 and computed its square: 16
[cse7b27@raman-pac MPI Programs]$
```

EXP 10 - CLOCK SYNCHRONIZATION

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

#define ROOT 0

int main(int argc, char** argv) {
    int num_processes, process_rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
    MPI_Comm_rank(MPI_COMM_WORLD, &process_rank);

    int local_time;
```

```
    if (process_rank == ROOT) {
        int i;
        printf("You have %d processes. Enter the logical clock values for each
process:\n", num_processes);
        fflush(stdout);

        for (i = 0; i < num_processes; i++) {
            if (i == ROOT) {
                printf("Enter the logical clock value for process %d: ", ROOT);
                fflush(stdout);
                scanf("%d", &local_time);
            } else {
                printf("Enter the logical clock value for process %d: ", i);
                fflush(stdout);
                int input_time;
                scanf("%d", &input_time);
                MPI_Send(&input_time, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            }
        }
    } else {
        MPI_Recv(&local_time, 1, MPI_INT, ROOT, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    }

    printf("Process %d has local time: %d\n", process_rank, local_time);
    fflush(stdout);

    int* local_times = NULL;
    if (process_rank == ROOT) {
        local_times = (int*)malloc(num_processes * sizeof(int));
    }

    MPI_Gather(&local_time, 1, MPI_INT, local_times, 1, MPI_INT, ROOT,
MPI_COMM_WORLD);

    if (process_rank == ROOT) {
        int i, sum = 0;
```

```
        printf("\nCoordinator (Process %d) has received the following local times:\n",
ROOT);
        fflush(stdout);

        for (i = 0; i < num_processes; i++) {
            printf("Process %d time: %d\n", i, local_times[i]);
            sum += local_times[i];
        }
        fflush(stdout);

        int average_time = sum / num_processes;
        printf("\nCoordinator calculated the average time: %d\n", average_time);
        fflush(stdout);

        int* adjustments = (int*)malloc(num_processes * sizeof(int));
        for (i = 0; i < num_processes; i++) {
            adjustments[i] = average_time - local_times[i];
            printf("Process %d should adjust its time by: %d\n", i, adjustments[i]);
            fflush(stdout);
        }

        for (i = 1; i < num_processes; i++) {
            MPI_Send(&adjustments[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }

        local_time += adjustments[ROOT];
        printf("Process %d adjusted its time by %d. New local time: %d\n", ROOT,
adjustments[ROOT], local_time);
        fflush(stdout);

        free(local_times);
        free(adjustments);
    } else {
        int adjustment;
        MPI_Recv(&adjustment, 1, MPI_INT, ROOT, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        local_time += adjustment;
```

```
        printf("Process %d adjusted its time by %d. New local time: %d\n",
process_rank, adjustment, local_time);
        fflush(stdout);
    }

    MPI_Finalize();
    return 0;
}
```

```
[cse7b27@raman-pac MPI Programs]$ mpicc ex10.c
[cse7b27@raman-pac MPI Programs]$ mpirun -np 4 a.out
You have 4 processes. Enter the logical clock values for each process:
Enter the logical clock value for process 0: 10
Enter the logical clock value for process 1: 11
Process 1 has local time: 11
Enter the logical clock value for process 2: 15
Enter the logical clock value for process 3: Process 2 has local time: 15
15
Process 1 adjusted its time by 1. New local time: 12
Process 2 adjusted its time by -3. New local time: 12
Process 0 has local time: 10

Coordinator (Process 0) has received the following local times:
Process 0 time: 10
Process 1 time: 11
Process 2 time: 15
Process 3 time: 15

Coordinator calculated the average time: 12
Process 0 should adjust its time by: 2
Process 1 should adjust its time by: 1
Process 2 should adjust its time by: -3
Process 3 should adjust its time by: -3
Process 0 adjusted its time by 2. New local time: 12
Process 3 has local time: 15
Process 3 adjusted its time by -3. New local time: 12
[cse7b27@raman-pac MPI Programs]$
```

EXP 11 - LEADER ELECTION

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

```
int main(int argc, char* argv[])
{
   MPI_Init(&argc, &argv);

   int rank, size;
   int leader;

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   int* uid = malloc(size * sizeof(int));
   int* token = malloc(size * sizeof(int));

   uid[rank] = rank * 100 + rank;

   if (rank == 0) {
      token[rank] = uid[rank];
   }

   if (rank != 0) {
      MPI_Recv(token, size, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      printf("Process %d received token from process %d\n", rank, rank - 1);
      token[rank] = uid[rank];
   }

   MPI_Send(token, size, MPI_INT, (rank + 1) % size, 0, MPI_COMM_WORLD);

   if (rank == 0) {
      MPI_Recv(token, size, MPI_INT, size - 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      printf("Process %d received token from process %d\n", rank, size - 1);

      int max = token[0];
      leader = 0;
      int i;
      for (i = 1; i < size; i++) {
```

```
            if (token[i] > max) {
                max = token[i];
                leader = i;
            }
        }

        MPI_Send(&leader, 1, MPI_INT, (rank + 1) % size, 1,
MPI_COMM_WORLD);
    }

    if (rank != 0) {
        MPI_Recv(&leader, 1, MPI_INT, rank - 1, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Process %d received leader information from process %d, Leader is
%d\n", rank, rank - 1, leader);
        MPI_Send(&leader, 1, MPI_INT, (rank + 1) % size, 1,
MPI_COMM_WORLD);
    }

    if (rank == 0) {
        MPI_Recv(&leader, 1, MPI_INT, size - 1, 1, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Process %d received leader information from process %d, Leader is
%d\n", rank, size - 1, leader);
    }

    free(uid);
    free(token);
    MPI_Finalize();
    return 0;
}
```

```
[cse7b27@raman-pac MPI Programs]$ mpicc ex11.c
[cse7b27@raman-pac MPI Programs]$ mpirun -np 4 a.out
Process 0 received token from process 3
Process 0 received leader information from process 3, Leader is 3
Process 1 received token from process 0
Process 1 received leader information from process 0, Leader is 3
Process 2 received token from process 1
Process 2 received leader information from process 1, Leader is 3
Process 3 received token from process 2
Process 3 received leader information from process 2, Leader is 3
[cse7b27@raman-pac MPI Programs]$
```