1. Demonstration of MPI_Send and MPI_Recv

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  int world_size;
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  int number;
  if (world_rank == 0) {
    number = -1; // Set the number to be sent
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("Process 0 sent number %d to process 1\n", number);
  } else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n", number);
  }
  MPI_Finalize();
  return 0;
}
```

OUTPUT:

```
Process 0 sent number -1 to process 1
Process 1 received number -1 from process 0
```

2. Demonstration of deadlock using point to point communication

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  // Initialize the MPI environment
  MPI_Init(&argc, &argv);
  // Get the rank of the process
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  int number;
  if (world_rank == 0) {
    // Process 0 attempts to receive from Process 1 before sending
    MPI_Recv(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("Process 0 completed exchange with process 1.\n");
  } else if (world_rank == 1) {
    // Process 1 attempts to receive from Process 0 before sending
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    printf("Process 1 completed exchange with process 0.\n");
  }
  // Finalize the MPI environment.
  MPI_Finalize();
  return 0;
}
```

3. Avoidance of deadlock by altering the call sequence

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
 MPI_Init(&argc, &argv);
 int world_rank;
 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 int number1,number2;
 if (world_rank == 0) {
   // Process 0 first sends a message to Process 1, then receives a message from Process 1
   number1 = -1;
   MPI_Send(&number1, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
   MPI_Recv(&number2, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   printf("Process 0 sent number %d and then received number %d from process 1\n", -1, number2);
 } else if (world_rank == 1) {
   // Process 1 first receives a message from Process 0, then sends a message to Process 0
   MPI_Recv(&number1, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   number2 = -2; // Set the number to be sent back to Process 0
   MPI_Send(&number2, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
   printf("Process 1 received number %d and then sent number %d to process 0\n", number1, -2);
 }
 MPI_Finalize();
 return 0;
}
```

OUTPUT:

```
Process 0 sent number -1 and then received number -2 from process 1
Process 1 received number -1 and then sent number -2 to process 0
```

4. Avoidance of deadlock by non-blocking calls

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
 MPI_Init(&argc, &argv);
 int world_rank;
 MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
 MPI_Request send_request, recv_request;
 MPI_Status status;
 int number1,number2;
 if (world_rank == 0) {
   // Process 0 initiates a non-blocking send and then a non-blocking receive
   number1 = -1;
   MPI_Isend(&number1, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &send_request);
   MPI_Irecv(&number2, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &recv_request);
   // Wait for the non-blocking send to complete
   MPI_Wait(&send_request, &status);
   // Do other work here while the receive is in progress
   // Wait for the non-blocking receive to complete
   MPI_Wait(&recv_request, &status);
   printf("Process 0 sent %d and received number %d from process 1\n", number1,number2);
 } else if (world_rank == 1) {
  number2 = -2;
  // Process 1 does the same in the reverse order
   MPI_Isend(&number2, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &send_request);
   MPI_Irecv(&number1, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &recv_request);
   // Wait for the non-blocking receive to complete
   MPI_Wait(&recv_request, &status);
   // Do other work here while the send is in progress
   // Wait for the non-blocking send to complete
   MPI_Wait(&send_request, &status);
   printf("Process 1 received %d and then sent number %d to process 0\n", number1,number2);
}
MPI_Finalize();
return 0;
}
```

OUTPUT:

```
Process 0 sent -1 and received number -2 from process 1
Process 1 received -1 and then sent number -2 to process 0
```

5. Avoidance of deadlock by using MPI_Sendrecv(bandwidth also)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  int send_number;
  int recv_number;
  MPI_Status status;
  if (world_rank == 0) {
    send_number = -1;
    // Process 0 sends a message to Process 1 and receives a message from Process 1
    MPI_Sendrecv(&send_number, 1, MPI_INT, 1, 0,
          &recv_number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
    printf("Process 0 sent number %d and received number %d from process 1\n", send_number,
recv_number);
  } else if (world_rank == 1) {
    send_number = -2;
    // Process 1 sends a message to Process 0 and receives a message from Process 0
    MPI_Sendrecv(&send_number, 1, MPI_INT, 0, 0,
          &recv_number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
    printf("Process 1 sent number %d and received number %d from process 0\n", send_number,
recv_number);
  }
  MPI_Finalize();
  return 0;
}
```

OUTPUT:

```
Process 0 sent number -1 and received number -2 from process 1
Process 1 sent number -2 and received number -1 from process 0
```

6. Demonstration of synchronization between the two phases of program

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  // Phase 1: Each process performs some work independently.
  // For demonstration purposes, we'll just print the process rank.
  printf("Process %d is performing the first phase of work\n", world_rank);
  // Simulate some work with a sleep.
  sleep(1);
  // Synchronize at the barrier to ensure all processes have finished Phase 1.
  MPI_Barrier(MPI_COMM_WORLD);
  // Phase 2: Each process knows that all others have completed Phase 1.
  printf("Process %d is performing the second phase of work\n", world_rank);
  // Simulate some work with a sleep.
  sleep(1);
  // Synchronize again if necessary. For example, before finalizing the program,
  // to ensure no process exits prematurely.
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```

OUTPUT:

```
Process 1 is performing the first phase of work
Process 0 is performing the first phase of work
Process 1 is performing the second phase of work
Process 0 is performing the second phase of work
```

7. Demonstration of Broadcast operation

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  int broadcast_value;
  if (world_rank == 0) {
    // The root process sets the broadcast value.
    broadcast_value = 123;
    printf("Process 0 broadcasting value %d\n", broadcast_value);
  }
  // Use MPI_Bcast to broadcast the value.
  MPI_Bcast(&broadcast_value, 1, MPI_INT, 0, MPI_COMM_WORLD);
  // Now all processes print the value received.
  printf("Process %d received value %d\n", world_rank, broadcast_value);
  MPI_Finalize();
  return 0;
}
```

OUTPUT:

```
Process 0 broadcasting value 123
Process 0 received value 123
Process 1 received value 123
```

8. Demonstration of MPI_Gather

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  int world_rank, world_size;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  // Each process creates a buffer that will hold its individual number.
  int send_number = world_rank;
  // The root process creates a receive buffer that will gather all the numbers.
  int *recv_buffer = NULL;
  if (world_rank == 0) {
    recv_buffer = (int *)malloc(world_size * sizeof(int));
    if (recv_buffer == NULL) {
      fprintf(stderr, "Unable to allocate memory for recv_buffer\n");
      MPI_Abort(MPI_COMM_WORLD, 1);
    }
  }
  // Gather all individual numbers to the root process.
  MPI_Gather(&send_number, 1, MPI_INT, recv_buffer, 1, MPI_INT, 0, MPI_COMM_WORLD);
  // The root process prints out the gathered numbers.
  if (world_rank == 0) {
    printf("Process 0 gathered numbers: ");
    for (int i = 0; i < world_size; i++) {
      printf("%d ", recv_buffer[i]);
    }
    printf("\n");
    free(recv_buffer);
  }
  MPI_Finalize();
  return 0;
}
```

OUTPUT:

```
Process 0 gathered numbers: 0 1
```

9. Demonstration of MPI_Scatter

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  int world_rank, world_size;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  // The root process initializes the send buffer.
  int *send_buffer = NULL;
  if (world_rank == 0) {
    send_buffer = (int *)malloc(world_size * sizeof(int));
    if (send_buffer == NULL) {
      fprintf(stderr, "Unable to allocate memory for send_buffer\n");
      MPI_Abort(MPI_COMM_WORLD, 1);
    }
    for (int i = 0; i < world_size; i++) {
      send_buffer[i] = i + 10; // Just example data.
    }
  }
  // Each process has a receive buffer for the incoming number.
  int recv_number;
  // Scatter the numbers from the root process to all processes.
  MPI_Scatter(send_buffer, 1, MPI_INT, &recv_number, 1, MPI_INT, 0, MPI_COMM_WORLD);
  // Each process prints the number it has received.
  printf("Process %d received number %d\n", world_rank, recv_number);
  // The root process frees the send buffer.
  if (world_rank == 0) {
    free(send_buffer);
  }
  MPI_Finalize();
  return 0;
}
```

OUTPUT:

```
Process 0 received number 10
Process 1 received number 11
```

10. Demonstration of MPI_Scatter and MPI_Gather in a single program

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  int world_rank, world_size;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  // Root process initializes the send buffer.
  int *send_buffer = NULL;
  if (world_rank == 0) {
    send_buffer = (int *)malloc(world_size * sizeof(int));
    for (int i = 0; i < world_size; i++) {
      send_buffer[i] = i; // Fill the send buffer with consecutive integers.
    }
  }
  // Each process has a receive buffer for the incoming number.
  int recv_number;
  // Scatter the numbers from the root process to all processes.
  MPI_Scatter(send_buffer, 1, MPI_INT, &recv_number, 1, MPI_INT, 0, MPI_COMM_WORLD);
  // Each process increments the received number.
  recv_number += 1;
  // Root process prepares the gather buffer.
  int *gather_buffer = NULL;
  if (world_rank == 0) {
    gather_buffer = (int *)malloc(world_size * sizeof(int));
  }
  // Gather the incremented numbers at the root process.
  MPI_Gather(&recv_number, 1, MPI_INT, gather_buffer, 1, MPI_INT, 0,
MPI_COMM_WORLD);
  // The root process prints the results after the gather.
  if (world_rank == 0) {
    printf("Root process has gathered incremented numbers:");
    for (int i = 0; i < world_size; i++) {
      printf(" %d", gather_buffer[i]);
    }
    printf("\n");
    free(send_buffer);
    free(gather_buffer);
  }
  MPI_Finalize();
  return 0;
}
```

OUTPUT:

Root process has gathered incremented numbers: 1 2

11. Demonstration of MPI_Reduce(MPI_MAX,MPI_MIN,MPI_SUM,MPI_PROD)

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  int world_rank, world_size;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  // Each process creates a random number.
  srand(world_rank); // Seed the random number generator to get different numbers for each process
  int my_number = rand() % 100; // Each process picks a random number between 0 and 99
  printf("Process %d generated number %d\n", world_rank, my_number);
  int max_value, min_value, sum_value, prod_value;
  // Reduce all of the local numbers into the max_value on the root process
  MPI_Reduce(&my_number, &max_value, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
  // Reduce all of the local numbers into the min_value on the root process
  MPI_Reduce(&my_number, &min_value, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);
  // Reduce all of the local numbers into the sum_value on the root process
  MPI_Reduce(&my_number, &sum_value, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  // Reduce all of the local numbers into the prod_value on the root process
  MPI_Reduce(&my_number, &prod_value, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);
  if (world_rank == 0) {
    printf("The maximum number is %d\n", max_value);
    printf("The minimum number is %d\n", min_value);
    printf("The sum of all numbers is %d\n", sum_value);
    printf("The product of all numbers is %d\n", prod_value);
  }
  MPI_Finalize();
  return 0;
}
```

OUTPUT:

```
Process 1 generated number 83
Process 0 generated number 83
The maximum number is 83
The minimum number is 83
The sum of all numbers is 166
The product of all numbers is 6889
```

12. Demonstration of MPI_Allreduce(MPI_MAX,MPI_MIN,MPI_SUM,MPI_PROD)

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
  MPI_Init(&argc, &argv);
  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  // Each process generates a random number.
  srand(time(NULL) + world_rank); /* Seed the random number generator to get different numbers
for each process*/
  int my_number = rand() % 100; // Each process picks a random number between 0 and 99
  printf("Process %d generated number %d\n", world_rank, my_number);
  int max_value, min_value, sum_value, prod_value;
  // Allreduce all of the local numbers into the max_value for every process
  MPI_Allreduce(&my_number, &max_value, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
  // Allreduce all of the local numbers into the min_value for every process
  MPI_Allreduce(&my_number, &min_value, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
  // Allreduce all of the local numbers into the sum_value for every process
  MPI_Allreduce(&my_number, &sum_value, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
  // Allreduce all of the local numbers into the prod_value for every process
  MPI_Allreduce(&my_number, &prod_value, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);
  // Each process prints the results after the allreduce.
  printf("Process %d reports:\n", world_rank);
  printf(" Maximum number: %d\n", max_value);
  printf(" Minimum number: %d\n", min_value);
  printf(" Sum of all numbers: %d\n", sum_value);
  printf(" Product of all numbers: %d\n", prod_value);
  MPI_Finalize();
  return 0;
}
```

OUTPUT:

```
Process 0 generated number 68
Process 1 generated number 60

Process 1 reports:
 Maximum number: 68
 Minimum number: 60
 Sum of all numbers: 128
 Product of all numbers: 4080

Process 0 reports:
 Maximum number: 68
 Minimum number: 60
 Sum of all numbers: 128
 Product of all numbers: 4080
```