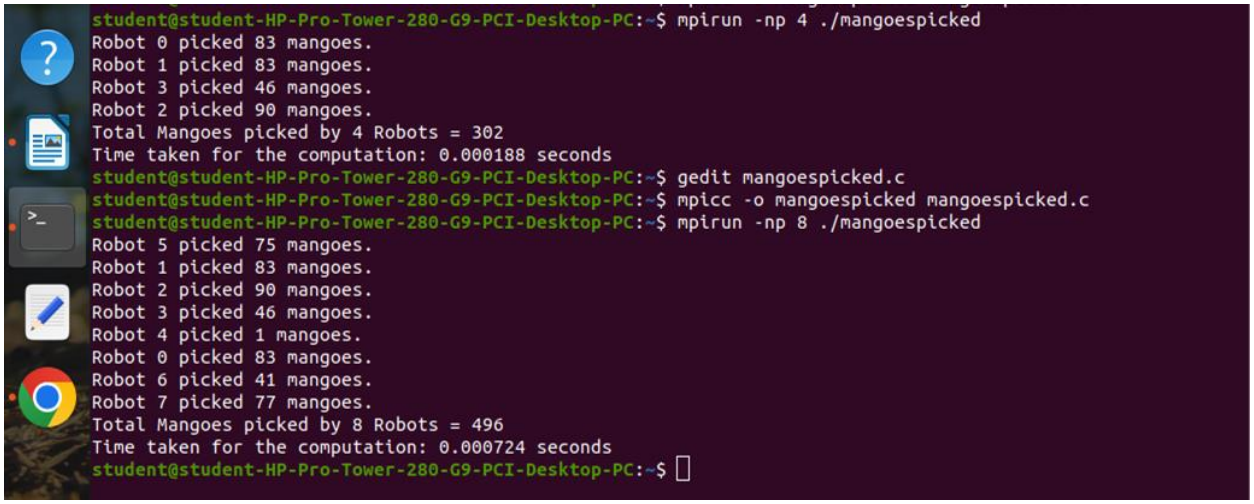
6th

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char** argv) {
   int rank, numproc;
   int sum = 0;
   int total_sum = 0;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numproc);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   srand(rank);
   sum = rand() % 100;
   printf("Robot %d picked %d mangoes.\n", rank, sum);
   // Start timing
   double start_time = MPI_Wtime();
   MPI_Reduce(&sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   // End timing
   double end_time = MPI_Wtime();
   if (rank == 0) {
      printf("Total Mangoes picked by %d Robots = %d\n", numproc, total_sum);
      printf("Time taken for the computation: %f seconds\n", end_time - start_time);
   }
   MPI_Finalize();
   return 0;
}
```

OUTPUT



7th

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char* argv[])
{
   int size, rank;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   float recvbuf, sendbuf[100];
   double start_time, end_time, time_taken;
   // Record start time
   start_time = MPI_Wtime();
   if (rank == 0) {
      int i;
      printf("Before Scatter : sendbuf of rank 0 : ");
      for (i = 0; i < size; i++) {
         srand(i);  // seeding random number generator for different values on each process
         sendbuf[i] = (float)(rand() % 1000) / 10;
         printf("%.1f ", sendbuf[i]);
      }
      printf("\nAfter Scatter :\n");
   }
   // Perform the scatter operation
   MPI_Scatter(sendbuf, 1, MPI_FLOAT, &recvbuf, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
   // Print the received data
   printf("rank= %d Recvbuf: %.1f\n", rank, recvbuf);
   // Record end time
   end_time = MPI_Wtime();
   // Calculate time taken for scatter operation
   time_taken = end_time - start_time;
   // Only rank 0 will print the total time taken
   if (rank == 0) {
      printf("Time taken for MPI_Scatter: %.6f seconds\n", time_taken);
   }
   MPI_Finalize();
}
```

OUTPUT

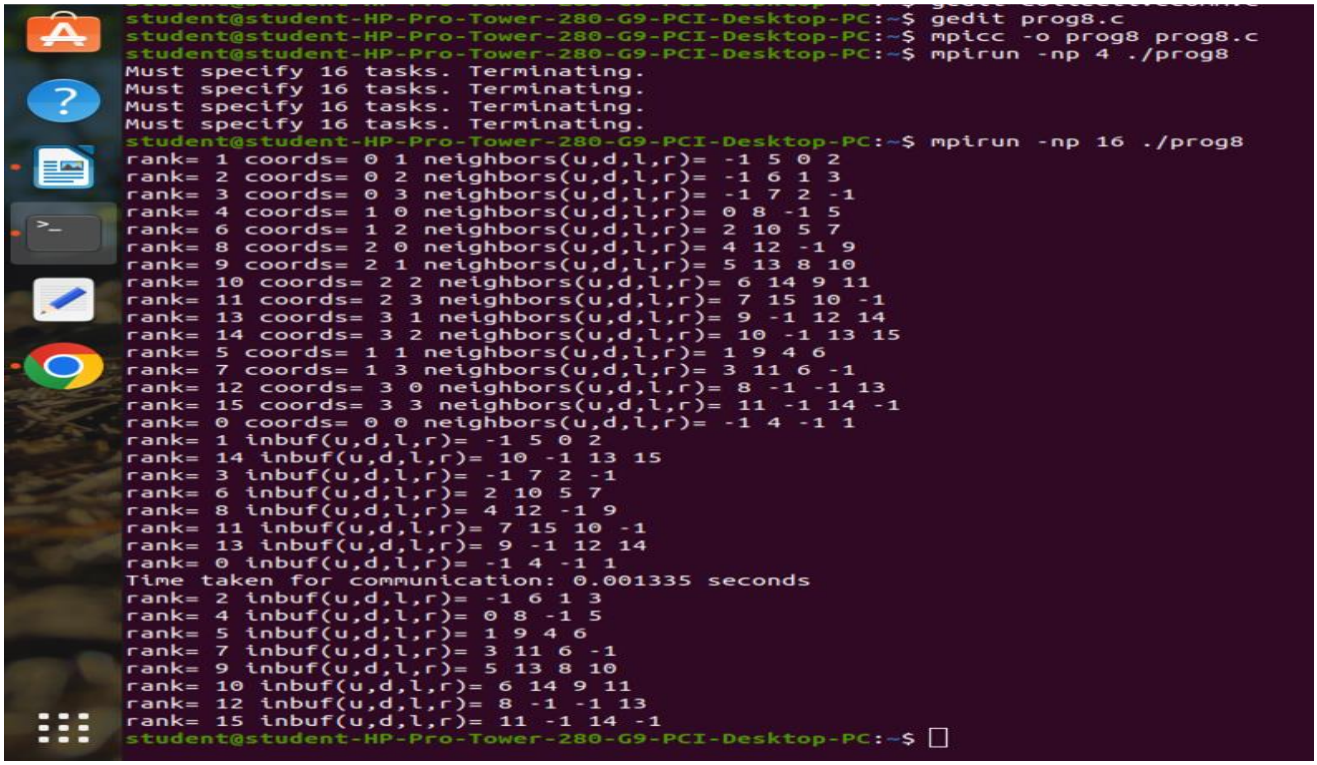
```
student@student-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ mpicc -o collectivecomm collectivecomm
student@student-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ mpirun -np 4 ./collectivecomm
Before Scatter : sendbuf of rank 0 : 38.3 38.3 29.0 74.6
After Scatter :
rank= 0 Recvbuf: 38.3
Time taken for MPI_Scatter: 0.000052 seconds
rank= 1 Recvbuf: 38.3
rank= 2 Recvbuf: 29.0
rank= 3 Recvbuf: 74.6
student@student-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ mpirun -np 8 ./collectivecomm
Before Scatter : sendbuf of rank 0 : 38.3 38.3 29.0 74.6 30.1 67.5 54.1 67.7
After Scatter :
rank= 0 Recvbuf: 38.3
Time taken for MPI_Scatter: 0.000071 seconds
rank= 1 Recvbuf: 38.3
rank= 2 Recvbuf: 29.0
rank= 4 Recvbuf: 30.1
rank= 6 Recvbuf: 54.1
rank= 5 Recvbuf: 67.5
rank= 3 Recvbuf: 74.6
rank= 7 Recvbuf: 67.7
student@student-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$
```

8th

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define SIZE 16
#define UP 0
#define DOWN 1
#define LEFT 2
#define RIGHT 3
int main(int argc, char* argv[]) {
   int numtasks, rank, source, dest, outbuf, i, tag = 1, inbuf[4] = {MPI_PROC_NULL,
MPI_PROC_NULL, MPI_PROC_NULL, MPI_PROC_NULL},
      nbrs[4], dims[2] = {4, 4}, periods[2] = {0, 0}, reorder = 0, coords[2];
   MPI_Request reqs[8];
   MPI_Status stats[8];
   MPI_Comm cartcomm;
   double start_time, end_time, time_taken;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
   if (numtasks == SIZE) {
      MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
      MPI_Comm_rank(cartcomm, &rank);
      MPI_Cart_coords(cartcomm, rank, 2, coords);
      MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
      MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);
      printf("rank= %d coords= %d %d neighbors(u,d,l,r)= %d %d %d %d\n", rank, coords[0],
coords[1], nbrs[UP], nbrs[DOWN], nbrs[LEFT], nbrs[RIGHT]);
      outbuf = rank;
      // Start timing the communication
      start_time = MPI_Wtime();
      // Initiate non-blocking communication
      for (i = 0; i < 4; i++) {
         dest = nbrs[i];
         source = nbrs[i];
         MPI_Isend(&outbuf, 1, MPI_INT, dest, tag, MPI_COMM_WORLD, &reqs[i]);
         MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag, MPI_COMM_WORLD, &reqs[i + 4]);
      }
      // Wait for all communication to complete
      MPI_Waitall(8, reqs, stats);
      // End timing the communication
      end_time = MPI_Wtime();
      // Calculate time taken for the communication
      time_taken = end_time - start_time;
      printf("rank= %d inbuf(u,d,l,r)= %d %d %d %d\n", rank, inbuf[UP], inbuf[DOWN],
inbuf[LEFT], inbuf[RIGHT]);
      // Print the time taken only from rank 0
      if (rank == 0) {
         printf("Time taken for communication: %.6f seconds\n", time_taken);
      }
   } else {
      printf("Must specify %d tasks. Terminating.\n", SIZE);
   }
```

```
    MPI_Finalize();
}
```

OUTPUT



9th

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<mpi.h>
int main(int argc, char* argv[])
{
        int numtasks, rank, rc, count, next, prev, sz, inmsg;
        MPI_Status Stat;
        time_t st, et;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
        sz = (numtasks / 2) * 2;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        st = clock();
        if (rank == 0) prev = sz - 1;
        else prev = rank - 1;
        if (rank == sz - 1) next = 0;
        else next = rank + 1;
        if (rank % 2 == 0 && rank < sz) {
                rc = MPI_Send(&rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD);
                rc = MPI_Recv(&inmsg, 1, MPI_INT, prev, 1, MPI_COMM_WORLD, &Stat);
        }
        else if (rank % 2 == 1 && rank < sz) {
                rc = MPI_Recv(&inmsg, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &Stat);
                rc = MPI_Send(&rank, 1, MPI_INT, next, 1, MPI_COMM_WORLD);
```

```
	}
	MPI_Barrier(MPI_COMM_WORLD);
	et = clock();
	if(rank==0) printf("Time taken by Blocking send/receive : %lf\n", (double)(et - st) /
CLOCKS_PER_SEC);
	MPI_Barrier(MPI_COMM_WORLD);
	MPI_Request reqs[2];
	MPI_Status stats[2];
	st = clock();
	if (rank == numtasks - 1) next = 0;
	else next = rank + 1;
	if (rank == 0) prev = numtasks - 1;
	else prev = rank - 1;
	MPI_Irecv(&inmsg, 1, MPI_INT, prev, 0, MPI_COMM_WORLD, &reqs[0]);
	MPI_Isend(&rank, 1, MPI_INT, next, 0, MPI_COMM_WORLD, &reqs[1]);
	MPI_Barrier(MPI_COMM_WORLD);
	et = clock();
	if (rank == 0) printf("Time taken by NonBlocking send/receive : %lf\n", (double)(et - st) /
CLOCKS_PER_SEC);
	MPI_Finalize();
}
```

OUTPUT

```
rank= 15 inbuf(u,d,l,r)= 11 -1 14 -1
student@student-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gedit prog9.c
student@student-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ mpicc -o prog9 prog9.c
student@student-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ mpirun -np 4 ./prog9
Time taken by Blocking send/receive : 0.000082
Time taken by NonBlocking send/receive : 0.000012
student@student-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$
```

10th

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<omp.h>
void main() {
	int n;
	printf("Enter the dimension of square matrices : ");
	scanf_s("%d", &n);
	int i = 0, j = 0, k = 0;
	int** arr1 = (int**)malloc(n * sizeof(int*));
	int** arr2 = (int**)malloc(n * sizeof(int*));
	int** res = (int**)malloc(n * sizeof(int*));
	omp_set_num_threads(64);
	#pragma omp parallel private(j)
	{
		#pragma omp for
		for (i = 0; i < n; i++) {
			srand(i);
			arr1[i] = (int*)malloc(n * sizeof(int));
			arr2[i] = (int*)malloc(n * sizeof(int));
			res[i] = (int*)malloc(n * sizeof(int));
			for (j = 0; j < n; j++) {
```

```
                        arr1[i][j] = rand() % 100;
                        arr2[i][j] = rand() % 100;
                }
        }
    }
    time_t st, et;
    st = clock();
    #pragma omp parallel private(j,k)
    {
        #pragma omp for
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                res[i][j] = 0;
                for (k = 0; k < n; k++)
                    res[i][j] += arr1[i][k] * arr2[k][j];
            }
        }
    }
    et = clock();
    printf("Time taken by parallel algorithm : %lf\n", (double)(et - st) / CLOCKS_PER_SEC);
    st = clock();
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            res[i][j] = 0;
            for (k = 0; k < n; k++)
                res[i][j] += arr1[i][k] * arr2[k][j];
        }
    }
    et = clock();
    printf("Time taken by Sequential algorithm : %lf\n", (double)(et - st) /
CLOCKS_PER_SEC);
}
```

OUTPUT