MPI

https://homepages.se.edu/kfrinkle/files/2018/01/MPI_scatter_example.cpp

COMPILE: mpicc sum_mpi.c -o sum_mpi

RUN : mpirun -np 3 ./sum_mpi

▼ TAG :

- The message tag is used to differentiate messages, in case rank A has sent multiple pieces of data to rank B.
- When rank B requests for a message the tag assists the receiving process in identifying the message.

▼ **Hello World** : Identify separate processes

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

int main(int argc, char *argv[])
{
  int id, num_proc, err;
  MPI_Status status;

  err = MPI_Init(&argc, &argv); /* Initialize MPI */
  if (err != MPI_SUCCESS) {
    printf("MPI initialization failed!\n");
    exit(1);
  }
  err = MPI_Comm_size(MPI_COMM_WORLD, &num_proc); /* Get nr of processes*/
  err = MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* Get id of this process */
  printf("Hello World! I'm process %i out of %i processes\n",id,num_procs);

  err = MPI_Finalize();
}
```

▼ Send and Recv : Slave processes (Rank (Id) and Size (No. of processes)) as Array Message

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

int main(int argc, char *argv[])
{
  const int tag = 42;        /* Message tag */
  int id, ntasks, source_id, dest_id, err, i;
  MPI_Status status;
  int msg[2]; /* Message array */

  err = MPI_Init(&argc, &argv); /* Initialize MPI */
  if (err != MPI_SUCCESS) {
    printf("MPI initialization failed!\n");
    exit(1);
  }
  err = MPI_Comm_size(MPI_COMM_WORLD, &ntasks); /* Get nr of tasks */
  err = MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* Get id of this process */
  if (ntasks < 2) {
    printf("You have to use at least 2 processors to run this program\n");
    MPI_Finalize();   /* Quit if there is only one processor */
    exit(0);
  }
if (id == 0) {  /* Process 0 (the receiver) does this */
    for (i=1; i<ntasks; i++) {
      err = MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, \
                     &status);          /* Receive a message */
      source_id = status.MPI_SOURCE; /* Get id of sender */
      printf("Received message %d %d from process %d\n", msg[0], msg[1], \
             source_id);
    }
  }
  else {    /* Processes 1 to N-1 (the senders) do this */
    msg[0] = id; /* Adding it's own identifier in the message */
    msg[1] = ntasks;        /* and total number of processes */
    dest_id = 0; /* Destination address */
    err = MPI_Send(msg, 2, MPI_INT, dest_id, tag, MPI_COMM_WORLD);
  }

  err = MPI_Finalize();         /* Terminate MPI */
  if (id==0) printf("Ready\n");
  exit(0);
  return 0;
}
```

▼ Time Code

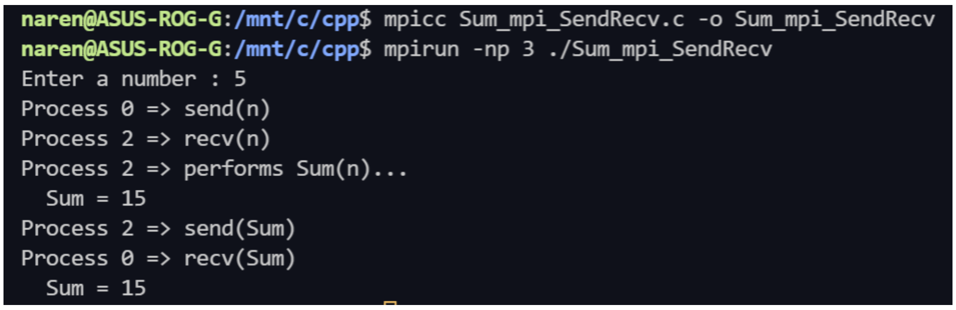
```
//Include files
#include<stdio.h>
#include <mpi.h>
int main( int argc, char** argv ) {
  int id, num_proc, err;
  double mytime;


  MPI_Status status;

  err = MPI_Init(&argc, &argv); /* Initialize MPI */
  if (err != MPI_SUCCESS) {
    printf("MPI initialization failed!\n");
    exit(1);
  }
  err = MPI_Comm_size(MPI_COMM_WORLD, &num_proc); /* Get nr of processes*/
  err = MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* Get id of this process */
  MPI_Barrier(MPI_COMM_WORLD);
  mytime = MPI_Wtime();
  work(d,num_proc, a, n); //Perform Work :
  mytime = MPI_Wtime() - mytime;
  printf("Timing from process %d is %lf seconds.\n",id,mytime);
  err = MPI_Finalize();         /* Terminate MPI */
  if (id==0) printf("Ready\n");
  exit(0);
  return 0;
}
```

▼ **Make each process perform a different task**

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

int main(int argc, char *argv[])
{
  int id, num_proc, err;
  MPI_Status status;
/* Create child processes, each of which has its own variables.
*From this point on, every process executes a separate copy
of this program. Each process has a different process ID,
* ranging from 0 to (num_procs - 1), and COPIES of all
* variables defined in the program. No variables are shared.
*/

  err = MPI_Init(&argc, &argv); /* Initialize MPI */
  if (err != MPI_SUCCESS) {
    printf("MPI initialization failed!\n");
    exit(1);
  }
  err = MPI_Comm_size(MPI_COMM_WORLD, &num_proc); /* Get nr of processes*/
  err = MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* Get id of this process */

  if(id=0) { /* Work for process 0 */ }
  else if(id=1) { /* Work for process 1 */ }
  else { /* Work for remaining processes */ }
  err = MPI_Finalize();
}
```

▼ Sum of N numbers (Send-Recv)

```
#include <stdio.h>
#include<string.h>
#include <mpi.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  int n,sum;
  char message[] = "START";
  int len = strlen(message);
  int id, ntasks,err,i;
  MPI_Status status;

  err = MPI_Init(&argc, &argv); /* Initialize MPI */
  if (err != MPI_SUCCESS) {
    printf("MPI initialization failed!\n");
    exit(1);
  }
  MPI_Comm_size(MPI_COMM_WORLD, &ntasks); /* Get nr of tasks */
  MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* Get id of this process */
  if (ntasks < 2) {
    printf("You have to use at least 2 processors to run this program\n");
    MPI_Finalize();   /* Quit if there is only one processor */
    exit(0);
  }
if (id == 0) {  // Process 0
    printf("Enter a number : ");
    scanf("%d", &n);
    printf("Process 0 => send(n)");

    MPI_Send(&n,1,MPI_INT,2,0,MPI_COMM_WORLD);
    }

if (id==2) {  // Process 2
     MPI_Recv(&n,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
     printf("\nProcess 2 => recv(n)");
     sum=0;
    printf("\nProcess 2 => performs Sum(n)...");
     for (i = 1; i <= n; ++i) {
        sum += i;
    }
    printf("\n  Sum = %d\n",sum);
    MPI_Send(&sum,1,MPI_INT,0,0,MPI_COMM_WORLD);
    printf("Process 2 => send(Sum)");
    }

if (id == 0) {
    MPI_Recv(&sum,1,MPI_INT,2,0,MPI_COMM_WORLD,&status);
    printf("\nProcess 0 => recv(Sum) \n  Sum = %d\n",sum);
  }

  MPI_Finalize();
  return 0;
}
```



```
naren@ASUS-ROG-G:/mnt/c/cpp$ mpicc Sum_mpi_SendRecv.c -o Sum_mpi_SendRecv
naren@ASUS-ROG-G:/mnt/c/cpp$ mpirun -np 3 ./Sum_mpi_SendRecv
Enter a number : 5
Process 0 => send(n)
Process 2 => recv(n)
Process 2 => performs Sum(n)...
  Sum = 15
Process 2 => send(Sum)
Process 0 => recv(Sum)
  Sum = 15
```

▼ Sum of N numbers (Scatter - Gather)

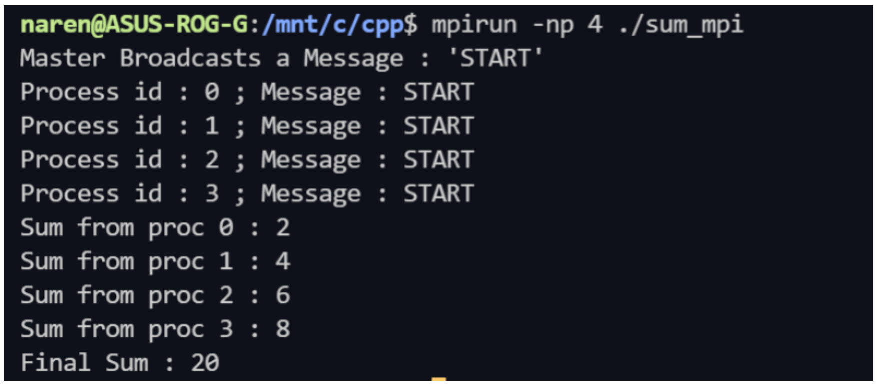
How to calculate average across multiple processes using MPI_Scatter and MPI_Gather?

You have some small problems with your code, namely unused variables, and variables not initialized. You should compile your code with some warning flags such as -Wall and -pedantic, among others.

https://stackoverflow.com/questions/65533005/how-to-calculate-average-across-multiple-processes-using-mpi-scatter-and-mpi-gat

arr[] = {1,1,2,2,3,3,4,4} ; chunk size = 2 ; No. of processes = 4

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

int main(int argc, char *argv[])
{
  int n,len,add;
  char message[] = "START";
  len = strlen(message);
  int id, ntasks, source_id, dest_id, err, i;
  MPI_Status status;
  int chunk = 2;
  int a[chunk];
    n = 8;
    int arr[] = {1,1,2,2,3,3,4,4};

  err = MPI_Init(&argc, &argv); /* Initialize MPI */
  if (err != MPI_SUCCESS) {
    printf("MPI initialization failed!\n");
    exit(1);


  }
  MPI_Comm_size(MPI_COMM_WORLD, &ntasks); /* Get nr of tasks */
  MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* Get id of this process */

   int out[ntasks];

  if (ntasks < 2) {
    printf("You have to use at least 2 processors to run this program\n");
    MPI_Finalize();   /* Quit if there is only one processor */
    exit(0);
  }
if (id == 0) {  // Process 0 : Master Process
    printf("Master Broadcasts a Message : '%s'\n",message);
    MPI_Bcast(&message, len, MPI_CHAR, 0,MPI_COMM_WORLD);
  }
    printf("Process id : %d ; Message : %s\n",id,message);
    MPI_Scatter(arr,chunk,MPI_INT,a,chunk,MPI_INT,0,MPI_COMM_WORLD);
    add = 0;
    for(int i=0;i<chunk;i++){
       add += a[i];
    }

    MPI_Gather(&add,1,MPI_INT, out, 1,MPI_INT,0,MPI_COMM_WORLD);

if (id == 0) { // Process 0 : Master Process
  int finalsum =0;
  for(int i=0;i<ntasks;i++)
  {
     printf("Sum from proc %d : %d\n",i,out[i]);
     finalsum += out[i];
  }


  printf("Final Sum : %d\n",finalsum);
  }
  MPI_Finalize();
  return 0;
}
```



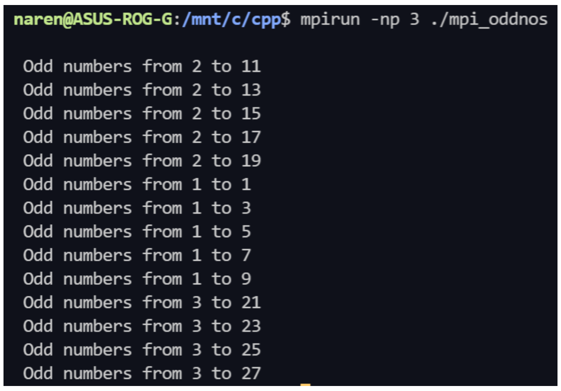
```
naren@ASUS-ROG-G:/mnt/c/cpp$ mpirun -np 4 ./sum_mpi
Master Broadcasts a Message : 'START'
Process id : 0 ; Message : START
Process id : 1 ; Message : START
Process id : 2 ; Message : START
Process id : 3 ; Message : START
Sum from proc 0 : 2
Sum from proc 1 : 4
Sum from proc 2 : 6
Sum from proc 3 : 8
Final Sum : 20
```

▼ Odd numbers (1 to num_proc*10)

If 1 to 100 ⇒ Assign 10 Process

(10 numbers is given to each process) 1⇒ 10 , 11⇒20

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char **argv)
{
int myid,numprocs,i;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD ,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD ,&myid);

for(i=myid*10+1;i<=myid*10+10;i++)
 {
 if(i%2==1)
 printf("\n Odd numbers from %d to %d ",myid+1,i);
 }
 MPI_Finalize();
}
```



```
naren@ASUS-ROG-G:/mnt/c/cpp$ mpirun -np 3 ./mpi_oddnos

 Odd numbers from 2 to 11
 Odd numbers from 2 to 13
 Odd numbers from 2 to 15
 Odd numbers from 2 to 17
 Odd numbers from 2 to 19
 Odd numbers from 1 to 1
 Odd numbers from 1 to 3
 Odd numbers from 1 to 5
 Odd numbers from 1 to 7
 Odd numbers from 1 to 9
 Odd numbers from 3 to 21
 Odd numbers from 3 to 23
 Odd numbers from 3 to 25
 Odd numbers from 3 to 27
```

▼ Even Numbers

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char **argv)
{
int myid,numprocs,i;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD ,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD ,&myid);

for(i=myid*10+1;i<=myid*10+10;i++)
 {
 if(i%2==0)
 printf("\n Even numbers from %d to %d ",myid+1,i);
 }
 MPI_Finalize();
}
```

▼ Array_SendRecv

MPI_Send(array,10,MPI_INT,1,tag,MPI_COMM_WORLD);

MPI_Recv (array,10,MPI_INT,0,tag,MPI_COMM_WORLD,&status);

Note :

- array is `array` and not `&array`
- All processes receiving data into an array need to have memory allocated for them too
- Send the size of the data first so the receiver knows how big to allocate the array
- Array must be allocated memory in Receiver side as well

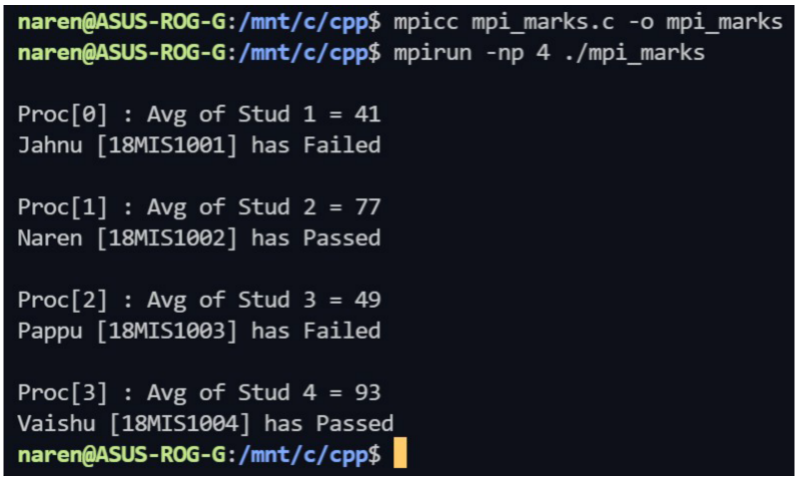
```
#include <stdio.h>
#include <string.h>
#include <mpi.h>
#include <stdlib.h>
      int main(int argc, char ** argv)
      {
       int * arr;
       int n;
       int tag=1;
       int size;
       int rank;
       MPI_Status status;
       MPI_Init (&argc,&argv);
```

```
        MPI_Comm_size (MPI_COMM_WORLD,&size);
        MPI_Comm_rank (MPI_COMM_WORLD,&rank);
        if (rank == 0)
        {
         printf("Enter the array size:");
         scanf("%d",&n);
         /*Send Array size,
         so the Receiver can allocate memory for the array to be received*/
         MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD);
         int array[n];
         printf("Array elements: ");
         for(int i=0;i<n;i++)
         {
             scanf("%d",&array[i]);
         }

         MPI_Send(array,n,MPI_INT,1,tag,MPI_COMM_WORLD);
        }
        if (rank == 1)
        {
         MPI_Recv (&n,1,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
         //int arr[n];
         arr = malloc (n * sizeof(int)); //Array of n elements
         MPI_Recv (arr,n,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
         printf("Array from proc %d : ",rank);
         for(int i=0;i<n;i++)
         {
             printf("%d ",arr[i]);
         }
        }
       MPI_Finalize();
       }
```

▼ Assume Proctors of VIT wants to create a program that helps them to calculate the average marks scored by their wards of CAT-1; Given inputs are register number, name, and Marks scored in any 5 courses; Write a MPI program which will implement the given scenario and also print the student is pass/ Fail.

NOTE:
• Consider No. of students = No. of processes
• Each row corresponds to the marks of each students
• Each process is conditioned to process the values of one row
• MPI_GATHER => To gather the values from each process and give it to process 0 (Master Process)
• Master process checks the condition and prints the result as Pass or Fail

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

int main(int argc, char *argv[])
{
  int n,len,sum,nsub,avg;

  int id, ntasks, source_id, dest_id, err, i;

  char *reg[]={"18MIS1001","18MIS1002","18MIS1085","18MIS1028"};
  char *name[]={"Jahnu","Naren","Pappu","Vaishu"};

    int x[4][5]={
            {32,35,42,50,48},
            {74,90,85,62,78},
            {76,75,32,21,43},
            {88,92,100,89,99}
                };

  MPI_Status status;
  err = MPI_Init(&argc, &argv); /* Initialize MPI */
  if (err != MPI_SUCCESS) {
    printf("MPI initialization failed!\n");
    exit(1);
  }
  MPI_Comm_size(MPI_COMM_WORLD, &ntasks); /* Get nr of tasks */
  MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* Get id of this process */

   int out[ntasks];

  if (ntasks < 2) {
    printf("You have to use at least 2 processors to run this program\n");
    MPI_Finalize();   /* Quit if there is only one processor */
    exit(0);
  }

for(int i=0;i<ntasks;i++){
    if(id==i) //Process Id == Row Id
    {
        sum = 0;
        for(int j=0;j<5;j++)
        {
            sum = sum + x[i][j];
        }
        avg = (sum/5);
    }
}

MPI_Gather(&avg,1,MPI_INT, out, 1,MPI_INT,0 /*Master Process*/,MPI_COMM_WORLD);

if (id == 0) { // Process 0 : Master Process
  for(int i=0;i<ntasks;i++)
    {
     printf("\nProc[%d] : Avg of Stud %d = %d\n",i,i+1,out[i]);
     if(out[i]>=50)
         printf("%s [%s] has Passed\n",name[i],reg[i]);
     else
         printf("%s [%s] has Failed\n",name[i],reg[i]);
    }
  }

  MPI_Finalize();
  return 0;
}
```



```
naren@ASUS-ROG-G:/mnt/c/cpp$ mpicc mpi_marks.c -o mpi_marks
naren@ASUS-ROG-G:/mnt/c/cpp$ mpirun -np 4 ./mpi_marks

Proc[0] : Avg of Stud 1 = 41
Jahnu [18MIS1001] has Failed

Proc[1] : Avg of Stud 2 = 77
Naren [18MIS1002] has Passed

Proc[2] : Avg of Stud 3 = 49
Pappu [18MIS1003] has Failed

Proc[3] : Avg of Stud 4 = 93
Vaishu [18MIS1004] has Passed
naren@ASUS-ROG-G:/mnt/c/cpp$
```

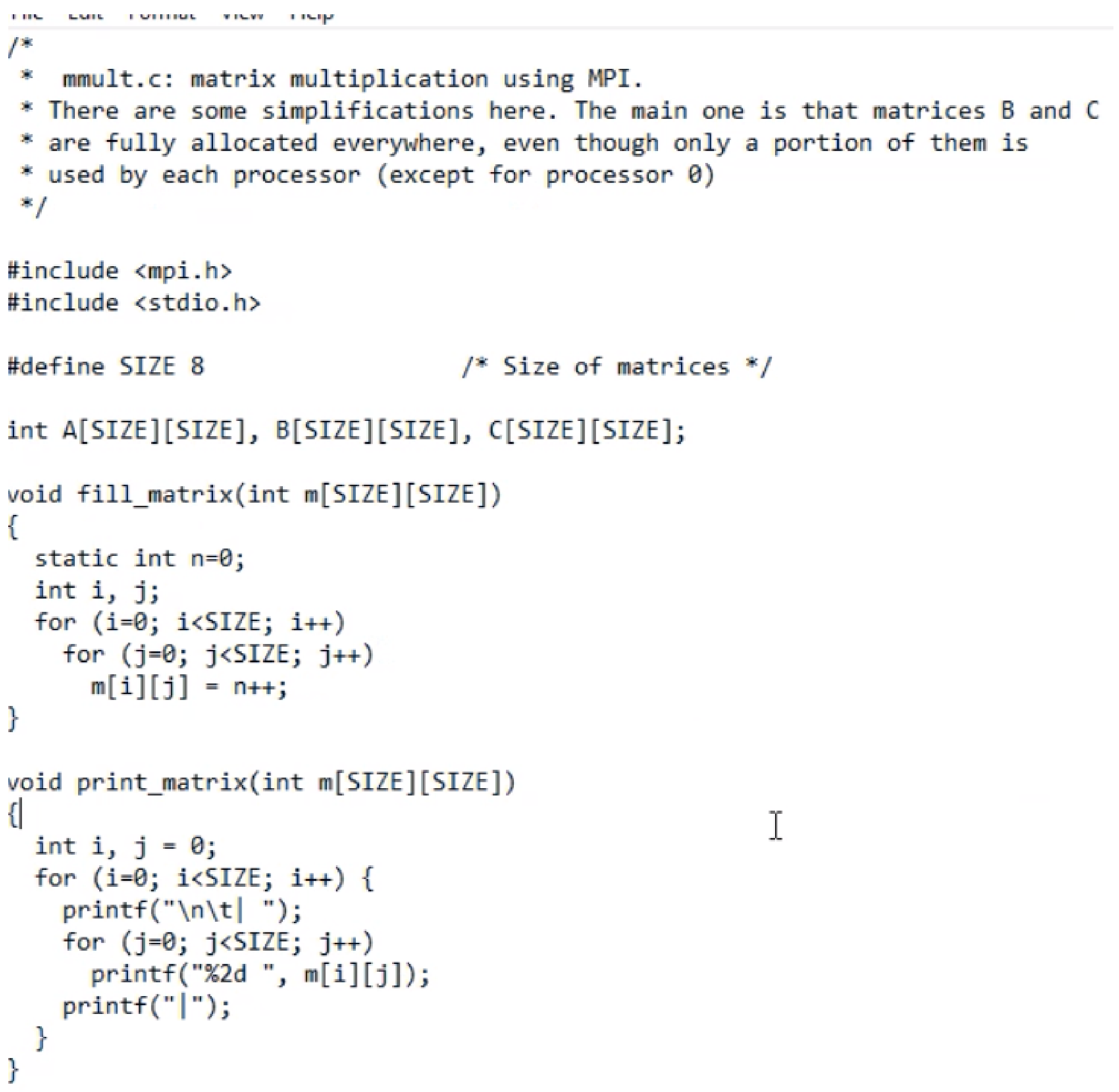
```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char** argv){
    int my_rank;
    int total_processes;
    int root = 0;
    int data[100];
    int data_loc[100];
    float final_res[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &total_processes);

    int input_size = 0;
    if (my_rank == 0){
       printf("Input how many numbers: ");
       scanf("%d", &input_size);

       printf("Input the elements of the array: ");
       for(int i=0; i<input_size; i++){
           scanf("%d", &data[i]);
       }
    }

    MPI_Bcast(&input_size, 1, MPI_INT, root, MPI_COMM_WORLD);

    int loc_num = input_size/total_processes;

    MPI_Scatter(&data, loc_num, MPI_INT, data_loc, loc_num, MPI_INT, root, MPI_COMM_WORLD);

    int loc_sum = 0;
    for(int i=0; i< loc_num; i++)
        loc_sum += data_loc[i];
    float loc_avg = (float) loc_sum / (float) loc_num;
    MPI_Gather(&loc_avg, 1, MPI_FLOAT, final_res, 1, MPI_FLOAT, root, MPI_COMM_WORLD);

    if(my_rank==0){
      float fin = 0;
      for(int i=0; i<total_processes; i++)
         fin += final_res[i];
      float avg = fin / (float) total_processes;
      printf("Final average: %f \n", avg);
    }
    MPI_Finalize();
    return 0;
}
```

A programmer wants to write a MPI program to compute sum of 'n' given natural numbers. In this process, he created 10 child processes and have divided the input array into small chunks in order to send each chunk to each process. Also the processes have to receive a common 'START' message from the master process to start with further computation. Illustrate the suitable NIPI ftnctions that the programmer can use to perform the said tasks.

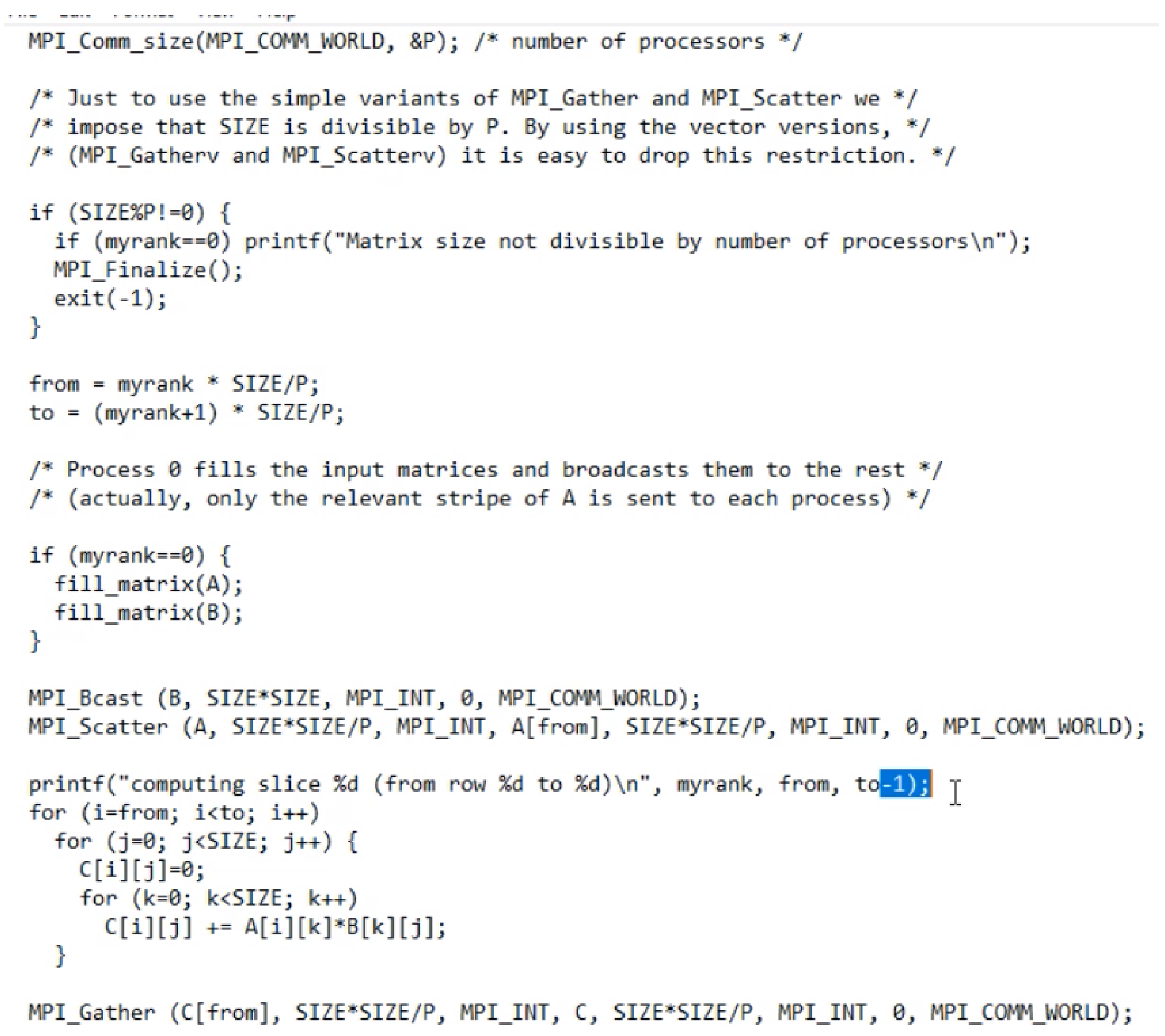
Sum of 2 arrays

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

```
#define max_rows 100000
#define send_data_tag 2001
#define return_data_tag 2002

int array[max_rows];
int array2[max_rows];

main(int argc, char **argv)
{
   long int sum, partial_sum;
   MPI_Status status;
   int my_id, root_process, ierr, i, num_rows, num_procs,
      an_id, num_rows_to_receive, avg_rows_per_process,
      sender, num_rows_received, start_row, end_row, num_rows_to_send;

   /* Now replicte this process to create parallel processes.
    * From this point on, every process executes a seperate copy
    * of this program */

   ierr = MPI_Init(&argc, &argv);

   root_process = 0;

   /* find out MY process ID, and how many processes were started. */

   ierr = MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
   ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

   if(my_id == root_process) {

      /* I must be the root process, so I will query the user
       * to determine how many numbers to sum. */

      printf("please enter the number of numbers to sum: ");
      scanf("%i", &num_rows);

      avg_rows_per_process = num_rows / num_procs;

      /* initialize an array */

      for(i = 0; i < num_rows; i++) {
         array[i] = i + 1;
      }

      /* distribute a portion of the bector to each child process */

      for(an_id = 1; an_id < num_procs; an_id++)
     {
         start_row = an_id*avg_rows_per_process + 1;
         end_row   = (an_id + 1)*avg_rows_per_process;

         if((num_rows - end_row) < avg_rows_per_process)
            end_row = num_rows - 1;

         num_rows_to_send = end_row - start_row + 1;

         ierr = MPI_Send( &num_rows_to_send, 1 , MPI_INT,
               an_id, send_data_tag, MPI_COMM_WORLD);

         ierr = MPI_Send( &array[start_row], num_rows_to_send, MPI_INT,
               an_id, send_data_tag, MPI_COMM_WORLD);
      }

      /* and calculate the sum of the values in the segment assigned
       * to the root process */

      sum = 0;
      for(i = 0; i < avg_rows_per_process + 1; i++) {
         sum += array[i];
      }

      printf("sum %i calculated by root process\n", sum);

      /* and, finally, I (the Master) collet the partial sums from the slave processes,
       * print them, and add them to the grand sum, and print it */

      for(an_id = 1; an_id < num_procs; an_id++) {

         ierr = MPI_Recv( &partial_sum, 1, MPI_LONG, MPI_ANY_SOURCE,
               return_data_tag, MPI_COMM_WORLD, &status);

         sender = status.MPI_SOURCE;

         printf("Partial sum %i returned from process %i\n", partial_sum, sender);

         sum += partial_sum;
      }

      printf("The grand total is: %i\n", sum);
   }

   else {

      /* I must be a slave process, so I must receive my array segment,
       * storing it in a "local" array, array1. */

      ierr = MPI_Recv( &num_rows_to_receive, 1, MPI_INT,
            root_process, send_data_tag, MPI_COMM_WORLD, &status);

      ierr = MPI_Recv( &array2, num_rows_to_receive, MPI_INT,
            root_process, send_data_tag, MPI_COMM_WORLD, &status);

      num_rows_received = num_rows_to_receive;

      /* Calculate the sum of my portion of the array */

      partial_sum = 0;
      for(i = 0; i < num_rows_received; i++) {
         partial_sum += array2[i];
      }

      /* and finally, send my partial sum to hte root process */

      ierr = MPI_Send( &partial_sum, 1, MPI_LONG, root_process,
            return_data_tag, MPI_COMM_WORLD);
   }
   ierr = MPI_Finalize();
}
```



```c
/*
 *  mmult.c: matrix multiplication using MPI.
 * There are some simplifications here. The main one is that matrices B and C
 * are fully allocated everywhere, even though only a portion of them is
 * used by each processor (except for processor 0)
 */

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

#define SIZE 8                  /* Size of matrices */

int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

void fill_matrix(int m[SIZE][SIZE])
{
  static int n=0;
  int i, j;
  for (i=0; i<SIZE; i++)
    for (j=0; j<SIZE; j++)
      m[i][j] = n++;
}

void print_matrix(int m[SIZE][SIZE])
{
  int i, j = 0;
  for (i=0; i<SIZE; i++) {
    printf("\n\t| ");
    for (j=0; j<SIZE; j++)
      printf("%2d ", m[i][j]);
    printf("|");
  }
}
```



```
MPI_Comm_size(MPI_COMM_WORLD, &P); /* number of processors */

/* Just to use the simple variants of MPI_Gather and MPI_Scatter we */
/* impose that SIZE is divisible by P. By using the vector versions, */
/* (MPI_Gatherv and MPI_Scatterv) it is easy to drop this restriction. */

if (SIZE%P!=0) {
  if (myrank==0) printf("Matrix size not divisible by number of processors\n");
  MPI_Finalize();
  exit(-1);
}

from = myrank * SIZE/P;
to = (myrank+1) * SIZE/P;

/* Process 0 fills the input matrices and broadcasts them to the rest */
/* (actually, only the relevant stripe of A is sent to each process) */

if (myrank==0) {
  fill_matrix(A);
  fill_matrix(B);
}

MPI_Bcast (B, SIZE*SIZE, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatter (A, SIZE*SIZE/P, MPI_INT, A[from], SIZE*SIZE/P, MPI_INT, 0, MPI_COMM_WORLD);

printf("computing slice %d (from row %d to %d)\n", myrank, from, to-1);
for (i=from; i<to; i++)
  for (j=0; j<SIZE; j++) {
    C[i][j]=0;
    for (k=0; k<SIZE; k++)
      C[i][j] += A[i][k]*B[k][j];
  }

MPI_Gather (C[from], SIZE*SIZE/P, MPI_INT, C, SIZE*SIZE/P, MPI_INT, 0, MPI_COMM_WORLD);
```