**7th April, 21**

**Steps to install MPI on Ubuntu**

$ sudo adduser mpi

$ passwd mpi

$ su mpi

$ sudo apt-get update

$ sudo rm /var/lib/dpkg/lock

$sudo rm /var/cache/apt/archives/lock

$ sudo apt-get install mpich

$ gedit hello.c

$mpicc hello.c –o h1

$mpirun -np 2 ./h1 –allow-run-as-root

**8th April,21**

**MPI Hello World Program**

#include <mpi.h>

#include <stdio.h>

int main(int argc, char\*\* argv) {

// Initialize the MPI environment

MPI\_Init(NULL, NULL);

// Get the number of processes

int world\_size;

MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

// Get the rank of the process

int world\_rank;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

// Get the name of the processor

char processor\_name[MPI\_MAX\_PROCESSOR\_NAME];

int name\_len;

MPI\_Get\_processor\_name(processor\_name, &name\_len);

// Print off a hello world message

printf("Hello world from processor %s, rank %d out of %d processors\n",

processor\_name, world\_rank, world\_size);

// Finalize the MPI environment.

MPI\_Finalize();

}

|  |
| --- |
| **Sum of Array Elements in MPI**  #include <mpi.h>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>    // size of array  #define n 10    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };    // Temporary array for slave process  int a2[1000];    int main(int argc, char\* argv[])  {        int pid, np,          elements\_per\_process,          n\_elements\_recieved;      // np -> no. of processes      // pid -> process id        MPI\_Status status;        // Creation of parallel processes      MPI\_Init(&argc, &argv);        // find out process ID,      // and how many processes were started      MPI\_Comm\_rank(MPI\_COMM\_WORLD, &pid);      MPI\_Comm\_size(MPI\_COMM\_WORLD, &np);        // master process      if (pid == 0) {          int index, i;          elements\_per\_process = n / np;            // check if more than 1 processes are run          if (np > 1) {              // distributes the portion of array              // to child processes to calculate              // their partial sums              for (i = 1; i < np - 1; i++) {                  index = i \* elements\_per\_process;                    MPI\_Send(&elements\_per\_process,                           1, MPI\_INT, i, 0,                           MPI\_COMM\_WORLD);                  MPI\_Send(&a[index],                           elements\_per\_process,                           MPI\_INT, i, 0,                           MPI\_COMM\_WORLD);              }                // last process adds remaining elements              index = i \* elements\_per\_process;              int elements\_left = n - index;                MPI\_Send(&elements\_left,                       1, MPI\_INT,                       i, 0,                       MPI\_COMM\_WORLD);              MPI\_Send(&a[index],                       elements\_left,                       MPI\_INT, i, 0,                       MPI\_COMM\_WORLD);          }            // master process add its own sub array          int sum = 0;          for (i = 0; i < elements\_per\_process; i++)              sum += a[i];            // collects partial sums from other processes          int tmp;          for (i = 1; i < np; i++) {              MPI\_Recv(&tmp, 1, MPI\_INT,                       MPI\_ANY\_SOURCE, 0,                       MPI\_COMM\_WORLD,                       &status);              int sender = status.MPI\_SOURCE;                sum += tmp;          }            // prints the final sum of array          printf("Sum of array is : %d\n", sum);      }      // slave processes      else {          MPI\_Recv(&n\_elements\_recieved, 1, MPI\_INT, 0, 0,                   MPI\_COMM\_WORLD,                   &status);            // stores the received array segment          // in local array a2          MPI\_Recv(&a2, n\_elements\_recieved,                   MPI\_INT, 0, 0,                   MPI\_COMM\_WORLD,                   &status);            // calculates its partial sum          int partial\_sum = 0;          for (int i = 0; i < n\_elements\_recieved; i++)              partial\_sum += a2[i];            // sends the partial sum to the root process          MPI\_Send(&partial\_sum, 1, MPI\_INT,                   0, 0, MPI\_COMM\_WORLD);      }        // cleans up all MPI state before exit of process      MPI\_Finalize();        return 0;  } |

**9th April, 21**

**MPI\_send / MPI\_Recv**

#include <stdio.h>

#include *<mpi.h>*

int main(int argc, char\*\* argv) {

int process\_Rank, size\_Of\_Cluster, message\_Item;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size\_Of\_Cluster);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &process\_Rank);

**if**(process\_Rank == 0){

message\_Item = 42;

MPI\_Send(&message\_Item, 1, MPI\_INT, 1, 1, MPI\_COMM\_WORLD);

printf("Message Sent: %d**\n**", message\_Item);

}

**else** **if**(process\_Rank == 1){

MPI\_Recv(&message\_Item, 1, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("Message Received: %d**\n**", message\_Item);

}

MPI\_Finalize();

**return** 0;

}

**Output:**

Message Sent: 42

Message Received: 42

**MPI\_status / MPI\_Tag in MPI**

How to receive a message without restricting the tag of the message. This application is meant to be run with 2 MPI processes: 1 sender and 1 receiver. It consists in the sender process sending a message to the receiver process, which will receive it without restricting the message tag during the reception operation. The receiver processes then concludes by printing the tag of the message obtained via the [MPI\_Status](https://www.rookiehpc.com/mpi/docs/mpi_status.php) collected.

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

**int** main(**int** argc, **char**\* argv[])

{

[MPI\_Init](https://www.rookiehpc.com/mpi/docs/mpi_init.php)(&argc, &argv);

// Check that 2 MPI processes are used.

**int** size;

[MPI\_Comm\_size](https://www.rookiehpc.com/mpi/docs/mpi_comm_size.php)([MPI\_COMM\_WORLD](https://www.rookiehpc.com/mpi/docs/mpi_comm_world.php), &size);

**if**(size != 2)

{

printf("This application is meant to be run with 2 MPI processes.\n");

[MPI\_Abort](https://www.rookiehpc.com/mpi/docs/mpi_abort.php)([MPI\_COMM\_WORLD](https://www.rookiehpc.com/mpi/docs/mpi_comm_world.php), EXIT\_FAILURE);

}

// Get my rank and do the corresponding job

**enum** role\_ranks { SENDER, RECEIVER };

**int** my\_rank;

[MPI\_Comm\_rank](https://www.rookiehpc.com/mpi/docs/mpi_comm_rank.php)([MPI\_COMM\_WORLD](https://www.rookiehpc.com/mpi/docs/mpi_comm_world.php), &my\_rank);

**switch**(my\_rank)

{

**case** SENDER:

{

// The "master" MPI process sends the message.

**int** buffer\_sent = 12345;

**int** tag = 67890;

printf("[MPI process %d] I send value %d with tag %d.\n", my\_rank, buffer\_sent, tag);

[MPI\_Ssend](https://www.rookiehpc.com/mpi/docs/mpi_ssend.php)(&buffer\_sent, 1, [MPI\_INT](https://www.rookiehpc.com/mpi/docs/mpi_int.php), RECEIVER, tag, [MPI\_COMM\_WORLD](https://www.rookiehpc.com/mpi/docs/mpi_comm_world.php));

**break**;

}

**case** RECEIVER:

{

// The "slave" MPI process receives the message.

**int** buffer\_received;

[MPI\_Status](https://www.rookiehpc.com/mpi/docs/mpi_status.php) status;

[MPI\_Recv](https://www.rookiehpc.com/mpi/docs/mpi_recv.php)(&buffer\_received, 1, [MPI\_INT](https://www.rookiehpc.com/mpi/docs/mpi_int.php), SENDER, [MPI\_ANY\_TAG](https://www.rookiehpc.com/mpi/docs/mpi_any_tag.php), [MPI\_COMM\_WORLD](https://www.rookiehpc.com/mpi/docs/mpi_comm_world.php), &status);

printf("[MPI process %d] I received value %d, with tag %d.\n", my\_rank, buffer\_received, status.[MPI\_TAG](https://www.rookiehpc.com/mpi/docs/mpi_tag.php));

**break**;

}

}

[MPI\_Finalize](https://www.rookiehpc.com/mpi/docs/mpi_finalize.php)();

**return** EXIT\_SUCCESS;

}