MPI

MPI es una Interfaz de Paso de Mensajes, se puede ver también como proceso de comunicación con ciertas restricciones, las cuales nos indican como y a quien se enviará el mensaje. Es el estándar para la comunicación entre los nodos que ejecutan un programa en un sistema de memoria distribuida. Las implementaciones en MPI consisten en un conjunto de bibliotecas de rutinas que pueden ser utilizadas en programas escritos en los lenguajes de programación C, C++, Fortran y Ada. La ventaja de MPI sobre otras bibliotecas de paso de mensajes, es que los programas que utilizan la biblioteca son portables (dado que MPI ha sido implementado para casi toda arquitectura de memoria distribuida), y rápidos, (porque cada implementación de la biblioteca ha sido optimizada para el hardware en la cual se ejecuta).

MPICC

Este comando se puede usar para compilar y vincular programas MPI escritos en C. Proporciona las opciones y las bibliotecas especiales que se necesitan para compilar y vincular programas MPI. La sintaxis de este comando sería la siguiente:

mpicc [*nombredelarchivo.c*] *−o* [*nombreaguardar*]

MPIrun

Mpirun es un comando que controla varios aspectos de la ejecución del programa en Open MPI. Mpirun utiliza un entorno de tiempo de ejecución abierto para iniciar trabajos. Podemos administrar los recursos.
La sintaxis de este comando se divide en dos tipos:
Programa Único:

mpirun [*opciones*] [*nombre − programa*]

Múltiple programa:

mpirun [*opciones*] [*nombre − programa*] : [*opciones*2] [*nombre − programa*2]

En este caso, podemos seleccionar en la parte de opciones el numero de procesadores que se desean utilizar, dependiendo del entorno (-np).
Para poder observar con mayor claridad, Se ha realizado como ejemplo un programa HolaMundo. Para compilar el programa utilizaremos MPIcc de la siguiente forma:

```
1 mpicc holamundo.c -o holamundo
```

Una vez compilado y que no haya marcado error alguno, procedemos a correr dicho programa mediante MPIrun

```
1 mpirun -np4 ./holamundo
```

El código que estamos compilando es el siguiente:

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char** argv) {
5      // Inicializa MPI
6      MPI_Init(NULL, NULL);
7
8      // Numero de procesos
9      int world_size;
10     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
11
12     // Obtener el rango del proceso
13     int world_rank;
```

```
14      MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
15
16       // Obtener el nombre del procesador
17      char processor_name[MPI_MAX_PROCESSOR_NAME];
18      int name_len;
19      MPI_Get_processor_name(processor_name, &name_len);
20
21     // Imprime un mensaje de hello world
22      printf("Hola mundo desde el proceso %s, rango %d de %d procesos\n",
23              processor_name, world_rank, world_size);
24
25      // Finaliza MPI
26      MPI_Finalize();
27  }
```

Y nos devolverá el siguiente resultado:

```
Hola mundo desde el proceso joiortega1-GL553VD, rango 0 de 4 procesos
Hola mundo desde el proceso joiortega1-GL553VD, rango 1 de 4 procesos
Hola mundo desde el proceso joiortega1-GL553VD, rango 2 de 4 procesos
Hola mundo desde el proceso joiortega1-GL553VD, rango 3 de 4 procesos
```

En este caso Particular, el rango hace referencia al número de procesador que es, y se obtiene con $world_rank$ mientras $world_size$ nos devuelve el total de procesos utilizados.

Implementaciones MPI

MPI cuenta con algunas funciones bastante útiles para la distribución o acceso a la informacion de los procesos, las mas usuales son las siguientes:

Scattering and Gathering.

Es la comunicación de información desde un proceso raíz hacía el resto de procesos, esta no es la misma, pues divide la información para distribuir una parte distinta a cada uno de ellos tal como se muestra en el siguiente ejemplo:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #include <mpi.h>
5  #include <assert.h>
6  float *create_rand_nums(int num_elements) {
7   float *rand_nums = (float *)malloc(sizeof(float) * num_elements);
8   assert(rand_nums != NULL);
9   int i;
10  for (i = 0; i < num_elements; i++) {
11    rand_nums[i] = (rand() / (float)RAND_MAX);
12  }
13  return rand_nums;
14 }
15 float compute_avg(float *array, int num_elements) {
16  float sum = 0.f;
```

```
17   int i;
18   for (i = 0; i < num_elements; i++) {
19     sum += array[i];
20   }
21   return sum / num_elements;
22 }
23 int main(int argc, char** argv) {
24   int num_elements_per_proc = atoi(argv[1]);
25   srand(time(NULL));
26   MPI_Init(&argc, &argv);
27   int world_rank;
28   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
29   int world_size;
30   MPI_Comm_size(MPI_COMM_WORLD, &world_size);
31   float *rand_nums = NULL;
32   if (world_rank == 0) {
33     rand_nums = create_rand_nums(num_elements_per_proc * world_size);
34   }
35   float *sub_rand_nums = (float *)malloc(sizeof(float) * num_elements_per_proc)
         ;
36   assert(sub_rand_nums != NULL);
37   MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT, sub_rand_nums,
38               num_elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);
39   float sub_avg = compute_avg(sub_rand_nums, num_elements_per_proc);
40   float *sub_avgs = (float *)malloc(sizeof(float) * world_size);
41   assert(sub_avgs != NULL);
42   MPI_Allgather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, MPI_COMM_WORLD)
         ;
43   float avg = compute_avg(sub_avgs, world_size);
44   printf("Promedio de todos los elementos del proceso %d es %f\n", world_rank,
         avg);
45   if (world_rank == 0) {
46     free(rand_nums);
47   }
48   free(sub_avgs);
49   free(sub_rand_nums);
50   MPI_Barrier(MPI_COMM_WORLD);
51   MPI_Finalize();
52 }
```

Broadcast.

Comunica información desde un proceso raíz hacia los demás procesos, siempre deberá ser la misma información y esta deberá llegar a cada uno de los procesos, es decir si el proceso A envía una valor X, ese valor le llegará al proceso B, C y D tal como en el siguiente ejemplo:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4 void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
```

```
5                 MPI_Comm communicator) {
6   int world_rank;
7   MPI_Comm_rank(communicator, &world_rank);
8   int world_size;
9   MPI_Comm_size(communicator, &world_size);
10  if (world_rank == root) {
11    // Si somos el proceso raíz, enviemos nuestros datos a todos
12    int i;
13    for (i = 0; i < world_size; i++) {
14      if (i != world_rank) {
15        MPI_Send(data, count, datatype, i, 0, communicator);
16      }
17    }
18  } else {
19    MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);
20  }
21 }
22
23 int main(int argc, char** argv) {
24   MPI_Init(NULL, NULL);
25
26   int world_rank;
27   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
28
29   int data;
30   if (world_rank == 0) {
31     data = 100;
32     printf("Proceso 0 Transmitiendo datos %d\n", data);
33     my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
34   } else {
35     my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
36     printf("El proceso %d recibio datos: %d del proceso raiz\n", world_rank,
         data);
37   }
38
39   MPI_Finalize();
40 }
```

Dando el siguiente resultado:

```
Proceso 0 envía: 10 a Proceso 1.
Proceso 0 envía: 10 a Proceso 2.
Proceso 0 envía: 10 a Proceso 3.
```

MPI_send y MPI_recv.

Las llamadas de envío y recepción de MPI funcionan cuando un proceso A empaqueta sus datos para ser enviados a un proceso B, este proceso B recibirá los datos empaquetados del proceso A para ser utilizados. Esta implementación será entre 2 o más procesos tal como se muestra en el siguiente código:

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

```
3  #include <stdlib.h>
4  int main(int argc, char** argv) {
5    MPI_Init(NULL, NULL);
6    int world_rank;
7    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
8    int world_size;
9    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
10   if (world_size < 2) {
11     fprintf(stderr, "World size must be greater than 1 for %s\n", argv[0]);
12     MPI_Abort(MPI_COMM_WORLD, 1);
13   }
14
15   int number;
16   if (world_rank == 0) {
17     number = -1;
18     MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
19   } else if (world_rank == 1) {
20     MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
21     printf("El proceso 1 recibió el Numero %d from process 0\n", number);
22   }
23   MPI_Finalize();
24 }
```

Dando el siguiente resultado:

```
Proceso 0 envía: -1 a Proceso 1.
```