Conceptos básicos de MPI

- **MPI → Message Passing Interface.**
- Es una especificación que define las propiedades de una librería de mensajes.
- Se basa en un modelo de programación paralelo basado en mensajes: la comunicación entre diferentes procesos es únicamente a través de mensajes.
- Es el standard de-facto para trabajar en clusters y sistemas distribuidos usando mensajes.
- **Historia**:
    - 1992 : Se forma el grupo de definición de estandares para transmisión de mensajes
    - 1994 : Se presenta la primera versión del estándar, MPI-1.0
    - 2008 : Se presenta la segunda versión del estándar, MPI-2.0
    - 2012 : Se aprueba la versión MPI-3.0
- La estructura básica de un programa MPI tiene 3 partes principales:
    1. Inicialización del ambiente MPI.
        Normalmente consiste de la función MPI_Init(&argc, &argv)
    2. Lógica del programa apoyada por MPI
        Aquí se usan las funciones específicas de MPI (MPI_Send, MPI_Recv, MPI_Bcast ...)
    3. Terminación del ambiente MPI
        Se llama a MPI_Finalize()

- **Comunicadores**: En MPI se definen objetos llamados 'comunicadores' (communicators), que definen los grupos de procesos que se comunican entre sí. El comunicador estándar se llama MPI_COMM_WORLD y engloba todos los procesos disponibles.
- **Rango/ID** : Dentro de cada comunicador cada proceso tiene asignado un rango, o identificador, que es el que lo identifica dentro del comunicador. Es un número del 0 a n-1, donde n es el número de procesos en el comunicador. Son usados para definir el destino o procedencia de los mensajes, así como para poder asignar lógica diferente a diferentes procesos. Para saber el número de rango de un proceso se usa MPI_Comm_rank

- **Programa básico de MPI**:

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

int main(argc,argv)
int argc;
char *argv[]; {
int  numtasks, rank, len, rc;
char hostname[MPI_MAX_PROCESSOR_NAME];

rc = MPI_Init(&argc,&argv);
if (rc != MPI_SUCCESS) {
  printf ("Error starting MPI program. Terminating.\n");
  MPI_Abort(MPI_COMM_WORLD, rc);
}

MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Get_processor_name(hostname, &len);
printf ("Number of tasks= %d My rank= %d Running on %s\n", numtasks,rank,hostname);
 /*******  Aca se hacen los cómputos auxiliados por MPI *******/
   MPI_Finalize();
}
```

Este programa se compilará con

```
$ mpicc prog.c -o prog
```

Y luego se correrá como:

```
$ mpirun -np N prog
```

Donde N es el número de procesos que queremos crear.

- **Tipos de datos:** Para poder comunicarse entre procesos, y para lograr que las implementaciones MPI sean independientes del hardware subyacente y los compiladores, MPI define tipos de datos. Estos tipos de datos pueden ser básicos, o una combinación de otros tipos de datos.
    - Tipos básicos: MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_BYTE, …
    - Arrays de otros tipos → MPI_Type_contiguous(count, oldtype, &newtype)
    - Vectores (con stride) → MPI_Type_vector(count, blocklength, stride, oldtype, &newtype)
    - Indexados → MPI_Type_indexed(count, blocklens[], offsets[], oldtype, &newtype)
    - Estructuras → MPI_Type_struct(count, blocklens[], offsets[], oldtypes[], &newtype)

- MPI-1.0 Define 2 tipos de comunicaciones:

+ **Comunicación punto a punto:** Este es el tipo de comunicación, o pase de mensajes, entre sólo 2 procesos. Se basa en funciones de envío y recepción de mensajes. Hacen uso de 'tags', que son simplemente números que identifican mensajes para poder identificar su uso o tipo. Se pueden usar definiciones especiales, MPI_ANY_SOURCE, MPI_ANY_TAG a la hora de recibir mensajes para obviar definir el origen o tag del mensaje a recibir.

Algunas de las funciones en este grupo son:

* **MPI_Send (&buf, count, datatype, dest, tag, comm)** -> Bloquea hasta que se manda el mensaje (no hasta que el receptor lo recibe).
* **MPI_SSend(&buf, count, datatype, dest, tag, comm)** -> Bloquea hasta que el receptor recibe el mensaje
* **MPI_RSend(&buf, count, datatype, dest, tag, comm)** -> Bloquea hasta que el receptor recibe el mensaje. Si no existe ningún receptor ya esperando con un receive, entonces falla.
* **MPI_ISend(&buf, count, datatype, dest, tag comm, &request)** -> No bloquea, se puede usar request para probar si el mensaje ya se envió con MPI_Test
* **MPI_IRecv(&buf, count, datatype, source, tag, comm, &request)** -> No bloquea, se puede usar request para probar si el mensaje ya se recibió con MPI_Test
* **MPI_Recv (&buf, count, datatype, source, tag, comm, &status)** -> Bloquea hasta recibir el mensaje.
* **MPI_Sendrecv(&sendbuf, sendcount, sendtype, dest, sendtag, &recvbuf&, recvcount, recvtype, source, recvtag, comm, &status)** -> Manda un mensaje y luego hace un receive. Bloquea hasta que el mensaje se haya mandado y un mensaje se haya recibido.

Ejemplo de comunicaciones punto a punto:

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

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  }

else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }

rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d \n",
       rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);


MPI_Finalize();
}
```

+ **Comunicación colectiva**: Es el tipo de comunicación donde un proceso manda información a muchos o un proceso recibe información de muchos, o para sincronizar grupos de procesos. Siempre involucran a TODOS los procesos de un comunicador. Solo pueden usar tipos de datos básicos, no derivados.

Algunas de las funciones en este grupo son:

* **MPI_Barrier(comm)** -> Punto de sincronización para todos los procesos del comunicador.
* **MPI_BCast(&buffer, count, datataype, root, comm)** -> Manda un mensaje del proceso root a todos los otros procesos del comunicador.
* **MPI_Reduce(&sendbuf, &recvbuf, count, datatype, op, root, comm)** -> Recibe un mensaje de todos los procesos de un comunicador y aplica una operación en root. Las operaciones predefinidas incluyen máximo, mínimo, suma, producto, y muchas más. Se pueden definir operaciones.

* **MPI_Scatter(&sendbuf, sendcount, senttype, &recvbuf, recvcount, recvtype, root, comm)** -> Distribuye mensajes distintos a cada proceso de un comunicador.
* **MPI_Gather(&sendbuf, sendcount, senttype, &recvbuf, recvcount, recvtype, root, comm)** -> Recibe un mensaje distinto de cada proceso en un comunicador.

Ejemplo de comunicaciones colectivas:

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
              MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
         recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Salida esperada:

rank= 0 Results: 1.000000 2.000000 3.000000 4.000000
rank= 1 Results: 5.000000 6.000000 7.000000 8.000000
rank= 2 Results: 9.000000 10.000000 11.000000 12.000000
rank= 3 Results: 13.000000 14.000000 15.000000 16.000000

- Características de MPI-2.0:

- Comunicación individual : permite operaciones con memoria compartida (put/get).
- Procesos dinámicos : permite la creación de nuevos procesos luego de haber inicializado MPI.
- Soporte para I/O paralelo.
- Operaciones colectivas entre subgrupos en un comunicador.

- Características de MPI-3.0:

- Operaciones colectivas no bloqueantes.
- Más operaciones individuales.
- Operaciones colectivas entre vecinos (para topologias virtuales)

Ejemplo de un programa que usa MPI para calcular el número pi usando el método de Monte Carlo (probabilistico).

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

int main(int argc, char* argv[]){
int i,id, np,N;
double x, y,double_N,eTime,sTime,pTime;
int lhit;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &np);

if( argc !=2){
      if (id==0){
            fprintf(stderr,"Incorrect number of args\n");
            fflush(stderr);
      }
      MPI_Abort(MPI_COMM_WORLD,1);
}

sscanf(argv[1], "%lf", &double_N);
N = lround(double_N);
MPI_Barrier(MPI_COMM_WORLD);
sTime = MPI_Wtime();
lhit = 0;
srand((unsigned)(time(0)));
int lN = N/np;

for(i = 0; i<lN;i++){
      x = ((double)rand())/((double)RAND_MAX);
      y = ((double)rand())/((double)RAND_MAX);
      if (((x*x) + (y*y)) <= 1) lhit++;
}

int hit=0;
MPI_Allreduce(&lhit,&hit,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
double est;
est = (hit*4)/((double)N);
MPI_Barrier(MPI_COMM_WORLD);

eTime = MPI_Wtime();
pTime = fabs(eTime - sTime);

if (id == 0) {
      printf("Number of Points Used:      %d\n",N);
      printf("Estimate of Pi:         %24.16f\n",est);
      printf("Elapsed Wall time:      %5.3e\n",pTime);
}

MPI_Finalize();
return 0;
}
```

Este algoritmo funciona creando puntos en un cuadrado. Dado que en un cuadrado de lado 2*R tiene area 4R^2, el circulo inscripto de radio R tiene area PI * R^2, el area del cuadrado cuadrado es 4/PI veces el area del circulo. Entonces si tomamos un punto (x,y) al azar en el cuadrado y vemos si pertenece o no al circulo (x^2+y^2 < R^2), tenemos una probabilidad de 4/PI de que el punto pertenezca al circulo. El algoritmo divide el trabajo de generar estos puntos al azar y calcular el valor (Puntos en el circulo)/(Puntos creados). Queremos calcular esto para N puntos, por lo cual cada uno de los np procesos hace la prueba para N/np puntos. Una vez que cada proceso termina su tarea, se llama a la función MPI_Allreduce que hace que se envien todos los resultados del grupo de procesos al proceso 0, se los suma y se obtiene una aproximación de PI a partir del mismo.

Bibliografía

- Recursos online en :
  - http://www.lam-mpi.org/tutorials/
  - https://computing.llnl.gov/tutorials/mpi/
  - http://www.mhpcc.edu/training/workshop/mpi/MAIN.html