Open-MPI

César Pedraza Bonilla

Universidad Nacional de Colombia
Departamento de Ingeniería de Sistemas e Industrial

capedrazab@unal.edu.co

6 de noviembre de 2019

Overview

Introducción.

- Es un conjunto de funciones para implementar paso de mensajes y operaciones complementarias.
- Especifica una interfaz *estándar* para C y fortran.
- Basado en modelo SPMD.
- La creación, inicialización y finalización de procesos depende directamente de la implementación.
- Las funciones retornan valores para conocer si se ejecutaron correctamente.
- Modos de comunicación: standard, synchronous, buffered, and ready.

Introducción.

- Groups: Un grupo es una representación local de un grupo de procesos MPI. Los grupos se representan mediante el tipo MPI_group.
- Communicators. Es un objeto local que representa la membresía de un proceso dentro de un grupo de procesos. Los *communicators* conforman un *group*.

Introducción.

Tipos de datos.

MPI_CHAR	signed char
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	signed short
MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	signed int
MPI_UNSIGNED	unsigned int
MPI_LONG	signed long
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Funciones MPI básicas.

- MPI_INIT : Inicializa entorno de ejecución MPI.
- MPI_FINALIZE : Finaliza entorno de ejecución MPI.
- MPI_COMM_SIZE : Determina no procesos del comunicador.
- MPI_COMM_RANK : Determina id. proceso en el comunicador.
- MPI_SEND : Envío básico mensaje.
- MPI_RECV : Recepción básica mensaje.

Iniciar y finalizar MPI:

```
1 int MPI_Init (int *argc, char ***argv)
2
3 int MPI_Finalize ( )
```

Funciones MPI básicas.


```
1
2  #include <mpi.h>
3
4  main( int argc, char** argv )
5  {
6      MPI_Init( &argc, &argv );
7
8      /* main part of the program */
9
10     /*
11      Use MPI function call depend on your data partitioning and the parallelization architecture
12     */
13
14     MPI_Finalize();
15 }
```

Funciones MPI básicas.

Comunicador = variable de tipo *MPI_Comm* = Grupo + Contexto.

- Grupo de procesos: es un conjunto de procesos que comparten un contexto.
- Contexto: Ambiente de paso de mensajes en el que se comunican los procesos.
- MPI_COMM_WORLD: es un comunicador por defecto para todos los procesos en ejecución.
- Un proceso puede pertenecer a diferentes comunicadores.

Funciones MPI básicas.

- int MPI_Comm_size (MPI_Comm_comm, int *size);
  Devuelve en *size* el número de procesos que pertenecen al comunicador *comm*.
- int MPI_Comm_rank (MPI_Comm_comm, int *rank);
  Devuelve en *rank* el identificador del proceso que lo llama en *comm*.

```
1  #include <stdio.h>
2  #include "mpi.h"
3    int main(int argc, char **argv ) {
4    int rank, size;
5    MPI_Init( &argc, &argv );
6      MPI_Comm_size( MPI_COMM_WORLD, &size );
7      MPI_Comm_rank( MPI_COMM_WORLD, &rank );
8      printf( "Hello world from process %d of %d\n", rank, size );
9    MPI_Finalize( );
10   return 0;
11 }
12
13 mpicc -o helloworld helloworld.c
14 mpirun -np 4 helloworld
15 Hello world from process 0 of 4
16 Hello world from process 3 of 4
17 Hello world from process 1 of 4
18 Hello world from process 2 of 4
```

Funciones MPI básicas.

- MPI_Send() Envía datos al proceso *dest* con etiqueta *tag* dentro del comunicador *comm*.

```
1 int MPI_Send ( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )
```

- MPI_Recv() Recibe datos datos.

```
1 int MPI_Recv ( void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm
        comm, MPI_Status *status )
```

```
1  MPI_Init( &argc, &argv );
2  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
3  MPI_Comm_size( MPI_COMM_WORLD, &size );
4  if (rank == 0) {
5    value=100;
6    MPI_Send (&value, 1, MPI_INT, 1, 0, MPI_COMM_WORLD );
7  }else
8    MPI_Recv ( &value, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
9
10 MPI_Finalize( );
```

Funciones MPI básicas.

Ejemplo.
Se desea enviar mensajes de un proceso a otro.

$P0 \longrightarrow P1 \longrightarrow P2 \longrightarrow P3$

Enviar: *rank* a *rank + 1*
Recibir: *rank* de *rank - 1*

Funciones MPI básicas.

Ejemplo.

```
1  #include <stdio.h>
2  #include "mpi.h"
3
4  int main(int argc, char **argv)
5  {
6    int rank, value, size;
7    MPI_Status status;
8    MPI_Init( &argc, &argv );
9    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
10   MPI_Comm_size( MPI_COMM_WORLD, &size );
11   do {
12     if (rank == 0) {
13       scanf( " %d", &value );
14       MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD );
15     }
16     else {
17       MPI_Recv( &value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status );
18       if (rank < size - 1)
19         MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD );
20     }
21     printf( "Process %d got %d\n", rank, value ); }
22   while (value >= 0);
23
24 MPI_Finalize( );
25 return 0;
26 }
```

Enviar y recibir.

- MPI_Sendrecv: permite enviar y recibir en una misma función.
- Importante para comunicaciones circulares.
- Combina MPI_Send y MPI_Recv.

```
1
2 int MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void
        *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm,
        MPI_Status *status )
```

Enviar y recibir.

- MPI_Sendrecv_replace: permite enviar y recibir en una misma función.
- Importante para comunicaciones circulares.
- Combina MPI_Send y MPI_Recv.
- Usa un único buffer.
- Send y Recv usan un mismo tipo de datos.

```
1 int MPI_Sendrecv_replace( void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source,
        int recvtag, MPI_Comm comm, MPI_Status *status )
```

Operaciones Colectivas.

Se usan con un comunicador y por tanto todos los procesos de ese comunicador.

- MPI_Barrier: Sincroniza todos los procesos.
- MPI_Broadcast: Envía un dato de un proc. al resto.
- MPI_Gather: Recolecta datos de todos los procesos a uno.
- MPI_Scatter: Reparte datos de un proceso a todos.
- MPI_Reduce: Realiza operaciones simples sobre datos distribuidos.
- MPI_Scan: Operación de reducción de prefijo.

Operaciones Colectivas.

MPI_Bcast() Distribuye datos de un proceso al resto de los procesos en un comunicador.

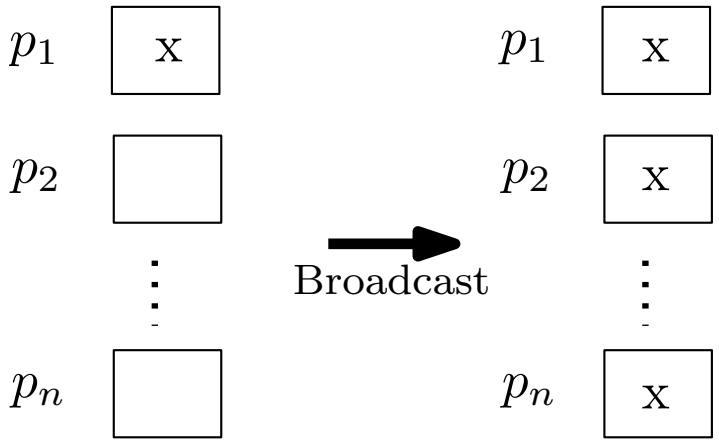

Figura: Single broadcast

```
1 MPI_Bcast(void *message, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

Operaciones Colectivas.


```
1  #define MSG_LENGTH 10
2  main (int argc, char *argv[])
3  {
4      int i, tag=0, tasks, iam, namelen, int root=0;
5      double x;
6      char message[MSG_LENGTH+2], processor_name[MPI_MAX_PROCESSOR_NAME];
7      MPI_Status status;
8      MPI_Init(&argc, &argv);
9      MPI_Comm_size(MPI_COMM_WORLD, &tasks);
10     MPI_Comm_rank(MPI_COMM_WORLD, &iam);
11     if (iam == 0) {
12         strcpy(message, "Hola Mundo");
13         MPI_Bcast(message, MSG_LENGTH, MPI_INT, root, MPI_COMM_WORLD);
14      } else {
15         MPI_Bcast(message, MSG_LENGTH, MPI_INT, root, MPI_COMM_WORLD);
16         MPI_Comm_rank(MPI_COMM_WORLD, &iam);
17         printf("\nnode  %d  %s ", iam, message);
18         MPI_Get_processor_name(processor_name, &namelen);
19         printf("processor  %s", processor_name); fflush(stdout);
20     }
21        MPI_Finalize();
22 }
```

Operaciones Colectivas.

MPI_Gather() . Cada proceso envía datos almacenados en el array *sendbuf* a *root*.

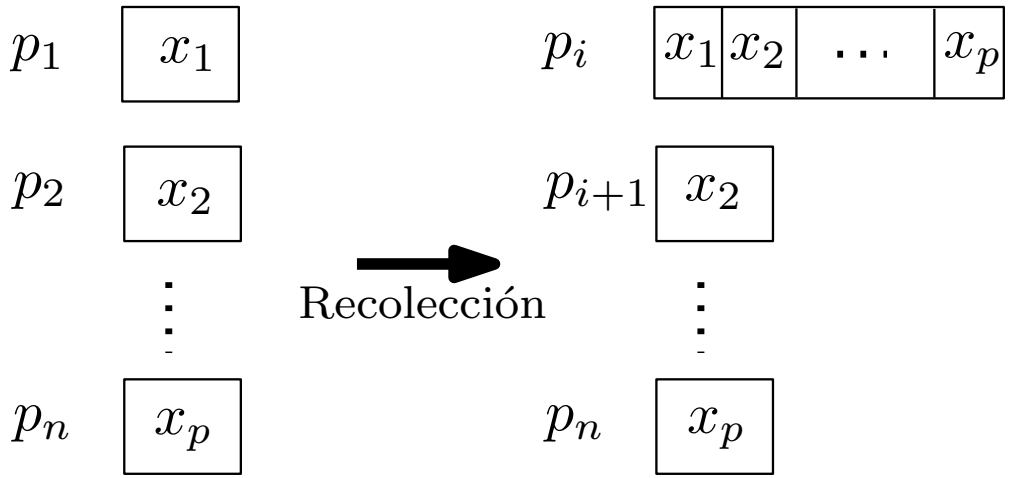

Figura: Gathering

```
1 int MPI_Gather ( void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount,
        MPI_Datatype recvtype, int root, MPI_Comm comm )
```

Operaciones Colectivas.


```
1  #define MSG_LENGTH 10
2  #define MAXTASKS 32
3  main (int argc, char *argv[])
4  {
5      int i, tag=0, tasks, iam, root=0, buff2send, buff2recv[MAXTASKS], namelen;
6      double x;
7      char processor_name[MPI_MAX_PROCESSOR_NAME];
8      MPI_Status status;
9      MPI_Init(&argc, &argv);
10     MPI_Comm_size(MPI_COMM_WORLD, &tasks);
11     MPI_Comm_rank(MPI_COMM_WORLD, &iam);
12     buff2send = iam;
13     if (iam == 0) {
14         MPI_Gather((void *)&buff2send, 1, MPI_INT, buff2recv, 1, MPI_INT, root, MPI_COMM_WORLD);
15         for(i=0; i < tasks; i++)
16             printf("  %i ", buff2recv[i]);
17      } else {
18         MPI_Comm_rank(MPI_COMM_WORLD, &iam);
19         MPI_Gather((void *)&buff2send, 1, MPI_INT, buff2recv, 1, MPI_INT, root, MPI_COMM_WORLD);
20         MPI_Comm_rank(MPI_COMM_WORLD, &iam);
21     }
22       MPI_Finalize();
23 }
```

Operaciones Colectivas.

MPI_Scatter()

```
1 int MPI_Scatter (void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount,
        MPI_Datatype recvtype, int root, MPI_Comm comm)
```

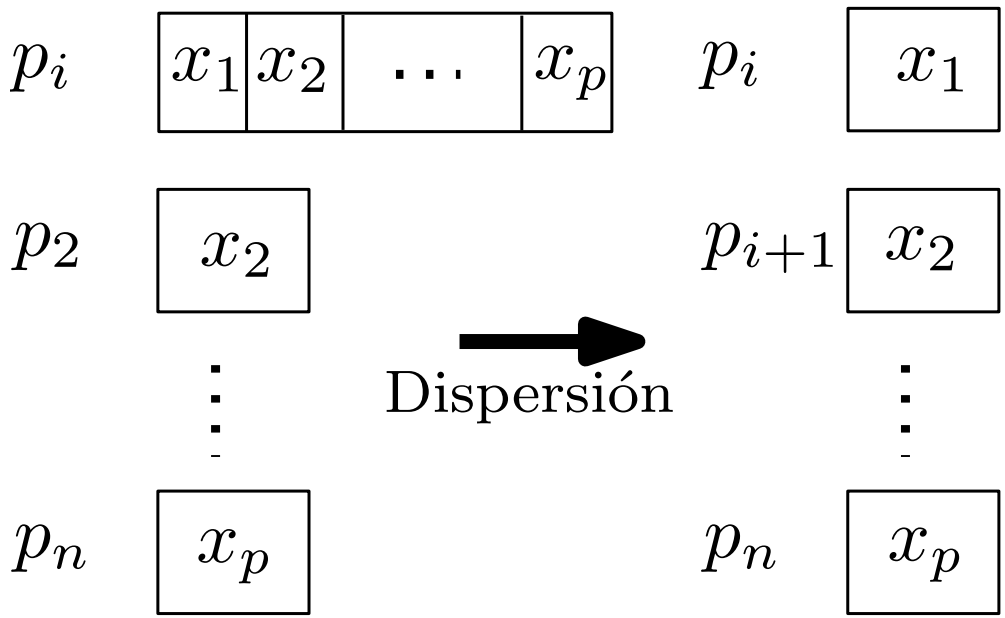

Figura: Scatter

Operaciones Colectivas.

MPI_Scatter()

```
1  #define MSG_LENGTH 10
2  #define MAXTASKS 32
3  main (int argc, char *argv[])
4  {
5      int i, tag=0, tasks, iam, root=0, buff2send[MAXTASKS], buff2recv, namelen;
6      double x;
7      char processor_name[MPI_MAX_PROCESSOR_NAME];
8      MPI_Status status;
9      MPI_Init(&argc, &argv);
10     MPI_Comm_size(MPI_COMM_WORLD, &tasks);
11     MPI_Comm_rank(MPI_COMM_WORLD, &iam);
12     if (iam == 0) {
13         for(i = 0; i < tasks; i++)
14             buff2send[i] = i;
15         MPI_Scatter((void *)buff2send, 1, MPI_INT, (void *)&buff2recv, 1, MPI_INT, root,
            MPI_COMM_WORLD);
16
17      } else {
18         MPI_Comm_rank(MPI_COMM_WORLD, &iam);
19         MPI_Scatter((void *)buff2send, 1, MPI_INT, (void *)&buff2recv, 1, MPI_INT, root,
            MPI_COMM_WORLD);
20         printf(" received: %i", buff2recv);
21     }
22        MPI_Finalize();
23 }
```

Operaciones Colectivas.

MPI_Reduce()

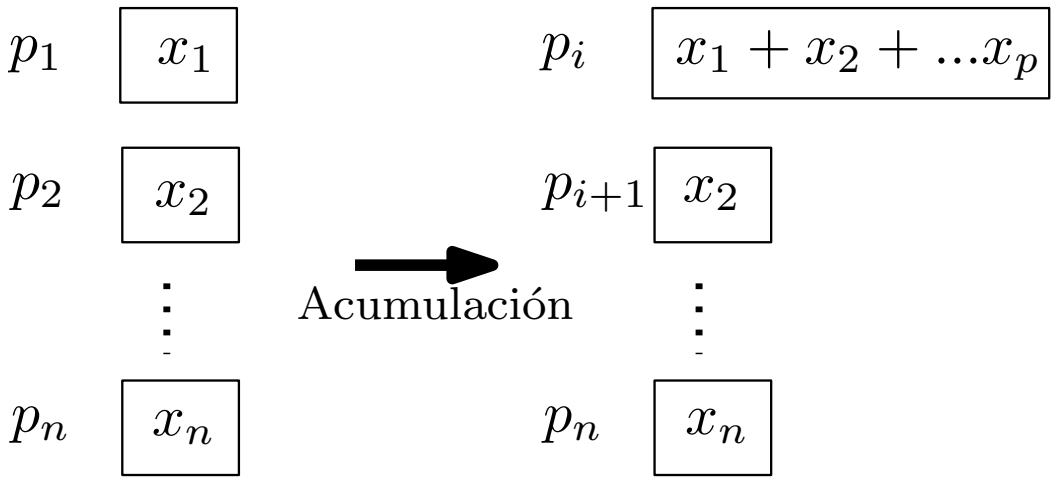

Figura: Reduce

```
1 MPI_Reduce(void *message, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root,
        MPI_Comm comm
```

Operaciones Colectivas.


```
1  int main(int argc, char *argv[])
2  {
3    int done = 0, n, myid, numprocs, I, rc;
4    double PI25DT = 3.14159265358979323846264643;
5    double mypi, pi, h, sum, x, a;
6    MPI_INIT(&argc, &argv);
7    MPI_COMM_SIZE(MPI_COMM_WORLD, &numprocs);
8    MPI_COMM_RANK(MPI_COMM_WORLD, &myid);
9    while (!done)
10   {
11     if (myid == 0)
12     {
13     printf("Enter the number of intervals: (0 quits) ");
14       scanf(" %d", &n);
15     }
16     MPI_BCAST(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
17     if (n == 0)
18   }
19
20     h = 1.0 / (double)n;
21   sum = 0.0;
22   for (i = myid + 1; i <= n; i += numprocs)
23   {
24     x = h * ((double)i - 0.5);
25     sum += 4.0 / (1.0 + x * x);
26   }
27   mypi = h * sum;
28   MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
29
30   if (myid == 0) printf("pi is approximately  %.16f, Error is  %.16f\n", pi, fabs(pi - PI25DT));
31
```

Operaciones Colectivas.