Programación Paralela en C con MPI

Miguel A. Astor y Ana Morales

EVI - CoNCISa 2016

Agenda

Comunicación con Paso de Mensajes

Paso de mensajes es un paradigma de comunicación entre procesos basado en dos primitivas:

`SEND(A, M)` Envía un mensaje `M` al proceso `A`.
`RECEIVE(A, M)` Recibe un mensaje `M` enviado por el proceso `A`

Ambas funciones pueden ser bloqueantes o no bloqueantes.

Llamadas bloqueantes

`SEND` Bloquea hasta que el receptor haya recibido el mensaje.

`RECEIVE` Bloquea mientras no se reciba un mensaje.

Llamadas no bloqueantes

`SEND` Retorna inmediatamente. El envío es asíncrono.

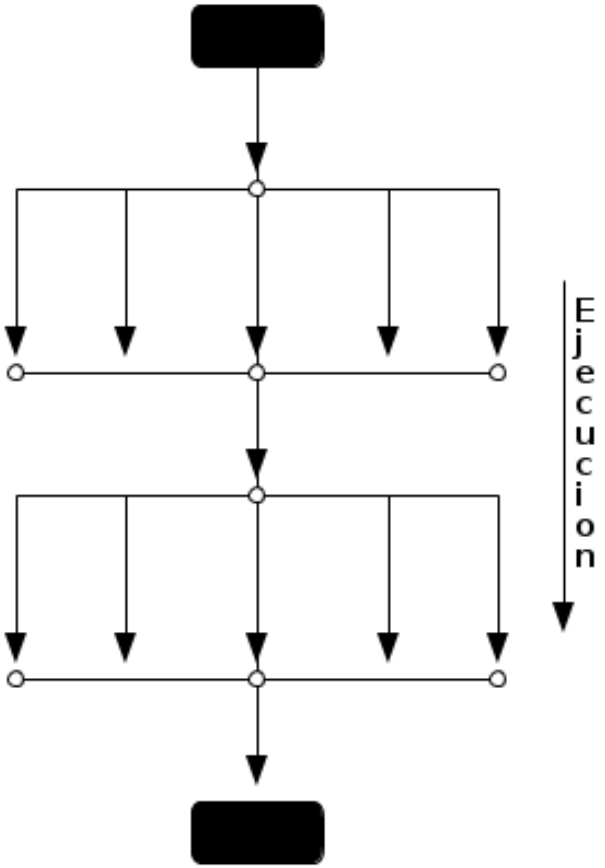
`RECEIVE` Dos semánticas posibles:

1. Retorna de inmediato si no hay mensajes disponibles.
2. Registra un *callback*.

El Patrón de Diseño Fork-Join

Fork-Join es un patrón de diseño para aplicaciones paralelas.

Un programa se compone de secciones seriales y secciones paralelas.
Al entrar a una sección paralela, la tarea a realizar se particiona y se reparte en procesos trabajadores.
Luego se acumulan los resultados.



¿Que es MPI?

MPI (*Message Passing Interface* - Interfaz de Paso de Mensajes) es una especificación de una biblioteca de paso de mensajes para los lenguajes de programación C/C++ y Fortran.
Existen múltiples implementaciones, una de las cuales es OpenMPI.

Funciones Fundamentales de MPI

Todo programa de MPI hace uso de estas funciones:

- `int MPI_Init(...)`
- `int MPI_Comm_size(...)`
- `int MPI_Comm_rank(...)`
- `int MPI_Finalize()`

Estas funciones se definen en la cabecera "*mpi.h*".

MPI Init

```
int MPI_Init(int *argc, char ***argv)
```

Inicializa MPI. Solo se debe llamar una vez.

Parámetros

`int *argc` Número de argumentos de linea de comandos.

`char ***argv` Argumentos de linea de comandos.

Retorno

- Código de error.

MPI Comm size

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Indica cuantos procesos hay en un grupo de comunicación.

Parámetros

`MPI_Comm comm` Grupo de comunicación. Usualmente `MPI_COMM_WORLD`.

`int *size` Número de procesos (SALIDA).

Retorno

- Código de error.

MPI Comm rank

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Indica cual es el rango de un proceso en un grupo de comunicación.

Parámetros

`MPI_Comm comm` Grupo de comunicación. Usualmente `MPI_COMM_WORLD`.

`int *rank` Rango del proceso (SALIDA).

Retorno

- Código de error.

MPI Finalize

```
int MPI_Finalize()
```

Termina el entorno de ejecución de MPI. Debe llamarse una única vez al final de todo proceso de MPI.

Retorno

- Código de error.

Esquema general de MPI

```
// Directivas #include
#include <mpi.h>

int main(int argc, char **argv) {
  // Declaración de variables.
  int size, rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  // Programa.
  MPI_Finalize();

  return EXIT_SUCCESS;
}
```

Comandos de MPI

Para compilar y ejecutar programas de MPI se utilizan estos comandos:

Compilación

mpicc *Wrapper* de GCC. Sigue la misma sintaxis.

Por ejemplo: `mpicc -o hello_mpi hello_mpi.c -lm`

Ejecución

mpirun Ejecuta un programa con el entorno de ejecución de MPI. Especifica cuantos sub-procesos utilizar, entre otras cosas.

Por ejemplo: `mpirun -np 9 hello_mpi`

Separación del programa en maestro y esclavos

Todo proceso creado por MPI tiene un rango (*rank*) único, obtenido de una serie creciente iniciada en 0.

Esquema general

El proceso de rango cero (0) es el maestro; todos los demás procesos son esclavos.

En código

```
int rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
  // Master
} else {
  // Slave.
}
```

Paso de mensajes

El paso de mensajes se lleva a cabo con estas funciones:

`int MPI_Send(...)` Envío.

`int MPI_Recv(...)` Recepción.

Ambas funciones son bloqueantes.

MPI Send

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

Parámetros

`void *buf` Apuntador a los datos a enviar.

`int count` Cantidad de datos a enviar.

`MPI_Datatype datatype` Tipo de los datos.

`int dest` Rango del proceso receptor.

`int tag` Identificador del mensaje.

`MPI_Comm comm` Grupo de comunicación. Usualmente `MPI_COMM_WORLD`.

MPI Recv

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

Parámetros

`void *buf` Apuntador a donde guardar los datos a recibir.

`int count` Cantidad de datos a recibir.

`MPI_Datatype datatype` Tipo de los datos.

`int source` Rango del proceso emisor.

`int tag` Identificador del mensaje.

`MPI_Comm comm` Grupo de comunicación. Usualmente `MPI_COMM_WORLD`.

`MPI_Status *status` Objeto de estado.

Tipos de datos

MPI define por lo menos los siguientes tipos de datos elementales:

Nombre de MPI	Tipo equivalente en C
MPI_CHAR ó MPI_SIGNED_CHAR *	char
MPI_SHORT *	short
MPI_INT *	int
MPI_LONG *	long int
MPI_LONG_LONG *	long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

* Estos tipos tienen equivalente UNSIGNED.

Finalización de programas

IMPORTANTE

Todo proceso creado con `MPI_Init` debe llamar a `MPI_Finalize` antes de terminar.

Ejercicio: Hello, MPI!

Requerimientos

- El maestro debe imprimir cuantos procesos activos hay.
- El maestro debe enviar la cadena Hello, Process %d! a cada esclavo, donde %d es el rango del esclavo.
- Cada esclavo debe imprimir la cadena recibida del maestro.
- Cada esclavo debe enviar la cadena Processor %d reporting for duty! al maestro donde %d es el rango del esclavo.
- El maestro debe imprimir las cadenas recibidas de los esclavos.

Ejercicio: Hello, MPI!

Ejercicio: Suma de un arreglo.

Requerimientos

- El maestro debe crear un arreglo de 1000 posiciones y llenarlas con el valor 9.
- El maestro debe enviar particiones contiguas del arreglo y enviarlas a los esclavos (asuma que 1000 es divisble exactamente entre la cantidad de esclavos).
- Los esclavos deben calcular la suma de los números de las particiones recibidas del maestro.
- Los esclavos deben enviar los resultados parciales al maestro.
- El maestro debe sumar los resultados parciales e imprimir el resultado final (9000).

Ejercicio: Suma de un arreglo.

Tópicos de Recepción de Mensajes

`MPI_Recv` puede recibir mensajes punto-a-punto de orígenes conocidos, o puede recibir mensajes de orígenes arbitrarios. Además, se pueden recibir más o menos datos de los esperados.

Parámetros

`int source` Puede ser un rango específico o el valor `MPI_ANY_SOURCE`.

`int tag` Puede ser un identificador específico o el valor `MPI_ANY_TAG`.

Mensajes arbitrarios

El objeto de estado `MPI_Status *status`, guarda información sobre quien envió el mensaje, que etiqueta tiene y cuantos datos contiene realmente. Si no se necesita, se puede utilizar el valor `MPI_STATUS_IGNORE`.

El objeto status

Una estructura de tipo `MPI_Status` contiene los siguientes campos:

`MPI_SOURCE` Rango del proceso emisor.

`MPI_TAG` Identificador del mensaje.

Es posible obtener la longitud de un mensaje utilizando la siguiente llamada:

```
int MPI_Get_count(MPI_Status * status,
  MPI_Datatype type, int * count)
```

Parámetros

`MPI_Status * status` Objeto de estado.

`MPI_Datatype type` Tipo de datos esperado.

`int * count` Cantidad de datos recibidos (SALIDA).

MPI Probe

Si se espera recibir un mensaje pero no se conoce su tamaño de antemano, se puede utilizar la función `MPI_Probe` para examinar el mensaje antes de recibirlo:

```
int MPI_Probe(int source, int tag,
      MPI_Comm comm, MPI_Status *status)
```

Parámetros

`int source` Rango del proceso emisor. Puede ser `MPI_ANY_SOURCE`.

`int tag` Identificador del mensaje a recibir. Puede ser `MPI_ANY_TAG`.

`MPI_Comm comm` Grupo de comunicación. Usualmente `MPI_COMM_WORLD`.

`MPI_Status *status` Objeto de estado.

Ejercicio: Hello, MPI! con status

Requerimientos

- El maestro debe imprimir cuantos procesos activos hay.
- El maestro debe enviar exactamente la cadena Hello, Process %d! a cada esclavo, donde %d es el rango del esclavo.
- Cada esclavo debe usar `MPI_Probe` para averiguar la cantidad de datos enviados por el maestro.
- Cada esclavo debe imprimir la cadena recibida del maestro.

Ejercicio: Hello, MPI! con status

Mensajes Broadcast

Con MPI es posible mandar un mismo mensaje a todos los procesos de un mismo grupo de comunicación con la función `MPI_Bcast`:

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

Parámetros

`void *buffer` Datos a enviar.

`int count` Cantidad de datos a enviar.

`MPI_Datatype datatype` Tipo de los datos.

`int root` Rango del proceso emisor. Se debe colocar la misma raiz en los receptores que en el emisor.

`MPI_Comm comm` Grupo de comunicación. Usualmente `MPI_COMM_WORLD`.

Ejercicio: Hello, MPI! con Broadcast

Requerimientos

- El maestro debe enviar la cadena Hello, Broadcast! a los esclavos con `MPI_Bcast`.
- Los esclavos deben escribir la cadena recibida.

Ejercicio: Hello, MPI! con Broadcast

Reduce

La función MPI_Reduce permite aplicar una operación a los resultados parciales calculados por los esclavos.

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

Parámetros

`void *sendbuf` Buffer de resultados parciales.

`void *recvbuf` Buffer acumulador.

`int count` Cantidad de elementos en los buffers.

`MPI_Datatype datatype` Tipo de datos de los buffers.

`MPI_Op op` Operación a aplicar.

`int root` Rango del proceso que realizará la reducción.

`MPI_Comm comm` Grupo de comunicación. Usualmente `MPI_COMM_WORLD`.

Operaciones de Reducción

`MPI_Reduce` puede aplicar las siguientes operaciones:

Operación	Acción
`MPI_MAX`	Calcula el máximo elemento
`MPI_MIN`	Calcula el mínimo elemento
`MPI_SUM`	Suma de elementos
`MPI_LAND`	AND lógico de todos los elementos
`MPI_BAND`	AND bit-a-bit de los elementos
`MPI_LOR`	OR lógico de todos los elementos
`MPI_BOR`	OR bit-a-bit de los elementos
`MPI_LXOR`	XOR lógico de todos los elementos
`MPI_BXOR`	XOR bit-a-bit de los elementos
`MPI_MAXLOC`	Máximo con ubicación
`MPI_MINLOC`	Mínimo con ubicación

Ejercicio: Suma de un arreglo con reducción.

Requerimientos

- Modificar el programa de suma de un arreglo para que calcule el resultado final utilizando `MPI_Reduce`.

Ejercicio: Suma de un arreglo.

Sincronización

MPI proporciona la función `MPI_Barrier` para sincronizar los procesos dentro de un grupo de comunicación. Cuando un proceso ejecuta `MPI_Barrier`, este se bloqueará hasta que todos los demás procesos de su grupo de comunicación ejecuten `MPI_Barrier`.

```
int MPI_Barrier(MPI_Comm comm)
```

Parámetros

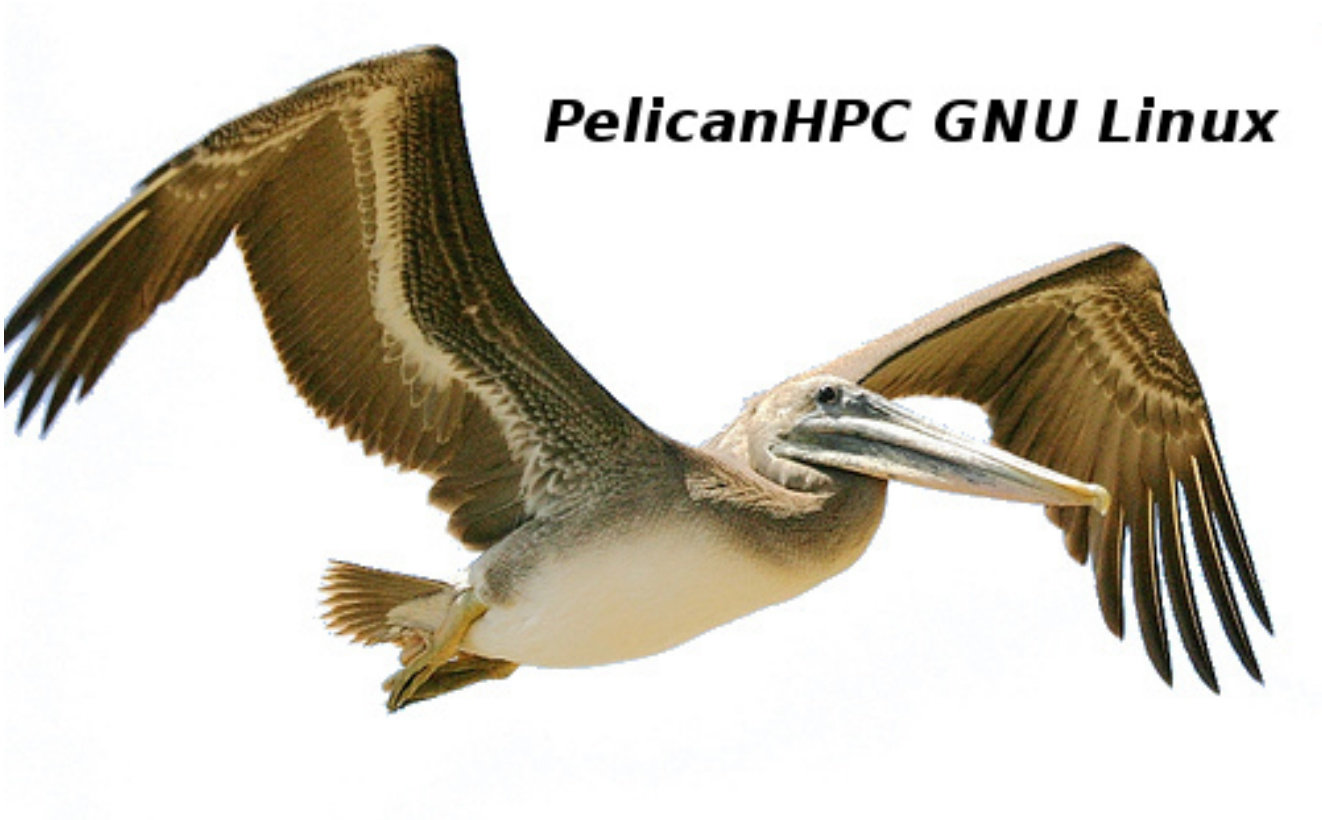
`MPI_Comm comm` Grupo de comunicación. Usualmente `MPI_COMM_WORLD`.

Otros mecanismos de sincronización

- Las funciones `MPI_Send` y `MPI_Recv` pueden usarse para sincronizar y coordinar pares de procesos.
- IMPORTANTE: Las funciones `MPI_Bcast` y `MPI_Reduce` implican la ejecución de `MPI_Barrier`.

PelicanHPC

http://pelicanhpc.awict.net/



Contactos

Prof. Miguel A. Astor

- miguel.astor@ciens.ucv.ve
- miguel.a.astor@ucv.ve

Profa. Ana Morales

- ana.morales@ciens.ucv.ve

¿Donde conseguir esta presentación y ejemplos?

- `https://github.com/miky-kr5/Presentations`

¿Preguntas?