



Programación Paralela - CC332

2021-I

José Fiestas

15/06/21

Universidad Nacional de Ingeniería
jose.fiestas@uni.edu.pe

Unidad 4: Comunicación y coordinación

Objetivos:

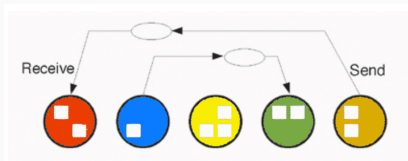
1. Pasos de Mensaje: MPI, Mensajes Punto a Punto, MPI, Comunicación Colectiva, Blocking vs non-blocking
2. Comunicacion Global, topologias
3. Memoria Compartida: OMP, Constructores y cláusulas, CUDA, optimizacion con GPUs
4. Programacion Hibrida

Message Passing Interface

Message Passing Interface

Comunicación (envío de mensajes) entre procesos:

- Utilizado en programación en paralelo (**MPI**) y orientada a objetos (C,C++,Fortran)
- No utiliza memoria compartida, sino espacio de memoria particionado en p nodos.
- Paralelismo explícito
- También llamado **Paralelismo del Especialista**

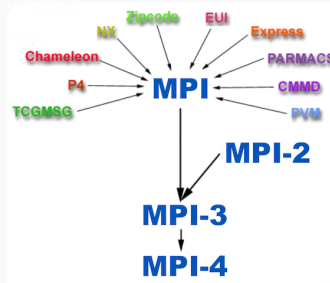


Message Passing Interface

Interfase para la programación usando el paradigma de paso de mensajes.
Aplicable a computadores de memoria compartida, distribuída e híbridos.

Se usa en C, C++, FORTRAN

Utiliza funciones y macros, y **paralelismo explícito**



Message Passing Interface

MPI produce una interfase standard. Fue creada en EEUU y Europa, vía el **MPI Forum** Luego de varios años de proposals, meetings y reviews, se creo el **standard MPI**

La interfase MPI permite portabilidad de código (escrito en C,C++,Fortran) a distintas arquitecturas

Ademas, soporta arquitecturas híbridas.

Lo que no es expl/'icito en MPI:

- Distribución de tareas en procesadores
- Creación de subprocesos durante ejecución
- Debugging
- Entrada/salida en paralelo

MPI es una librería . Un proceso MPI consiste en un código C++/Fortran, que se comunica con otros procesos llamando rutinas MPI.

Puede combinarse con otros métodos de comunicación pero no está totalmente definido para ello (e.g. OpenMP, problemas de sincronización)

Message Passing Interface

Se utilizan scripts para compilar programas MPI , que ya incluyen las correspondientes líneas

Language	Script Name	Underlying Compiler
C	<code>mpicc</code>	gcc
	<code>mpigcc</code>	gcc
	<code>mpiicc</code>	icc
	<code>mpigcc</code>	pgcc
C++	<code>mpicC</code>	g++
	<code>mpig++</code>	g++
	<code>mpiicpc</code>	icpc
	<code>mpigCC</code>	pgCC
Fortran	<code>mpif77</code>	g77
	<code>mpigfortran</code>	gfortran
	<code>mpiifort</code>	ifort
	<code>mpipgf77</code>	pgf77
	<code>mpipgf90</code>	pgf90

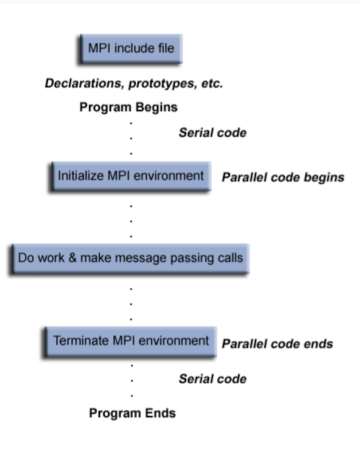
Message Passing Interface

Estructura
de un programa en MPI:

Requiere:

`#include<mpi.h>`

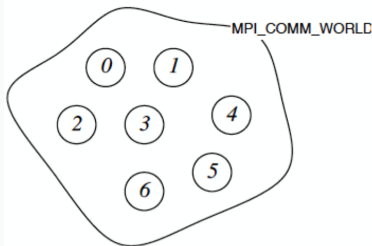
Cada proceso está
identificado con un número entero
(rank), que se inicia en cero



Message Passing Interface

MPI_COMM_WORLD:

MPI_Init define y llama **MPI_COMM_WORLD** para cada proceso. Que define la forma de comunicación entre estos. Este contiene una lista de procesos, que son numerados desde 0, y asignados a la variable **rank**



MPI Handles:

Son referencias a estructuras de data en MPI. Son variables de entorno en algunas directivas MPI, y pueden ser argumento de otras.

Ejemplos:

MPI_SUCCESS: un entero usado para detectar errores de código

MPI_COMM_WORLD: en C, un objeto de tipo MPI_Comm (un “communicator”). Representa un comunicador pre-definido conformado por todos los procesos.

Handles pueden ser copiados con las operaciones standar de asignación

MPI Errors:

En C, rutinas MPI retornan un entero, con un código de error. De ser detectado, el código aborta

```
int ierr;  
...  
ierr = MPI_Init (&argc, &argv);  
...
```

MPI_Init:

Inicializa MPI. Se llama al inicio y una sola vez en el programa

```
MPI_Init(&argc, &argv)
```

MPI_Comm_size:

Retorna el número total de procesos MPI

```
MPI_Comm_size(comm, &size)
```

MPI_Wtime:

Retorna el tiempo de reloj en segundos (double precision) en el proceso

```
MPI_Wtime()
```

MPI_Finalize:

Termina la ejecución MPI y limpia todas sus estructuras

```
MPI_Finalize()
```

MPI_Abort:

Termina todos los procesos MPI

```
MPI_Abort(comm, error)
```

MPI_Comm_rank:

Retorna el *rank* del proceso MPI

```
MPI_Comm_rank(comm, &rank)
```

MPI_Get_processor_name:

Retorna el nombre del procesador

```
MPI_Get_processor_name(&name, &lenght)
```

Ejemplo 01: "Hola mundo.^{en} MPI

```
1 # include <mpi.h>
2 # include <iostream>
3 using namespace std;
4 int main(int argc, char*argv[])
5 {
6     int rank, size;
7     MPI_Init(&argc,&argv);
8     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9     MPI_Comm_size(MPI_COMM_WORLD, &size);
10    cout<<"Hola"<<"soy"<<rank<<" de " <<size<<endl;
11    MPI_Finalize();
12 }
```

Compilación en C++:

```
mpiCC -o ejemplo.exe ejemplo.cpp
```

Ejecución en 4 procesadores:

```
mpiexec -np 4 ./ejemplo.exe
```

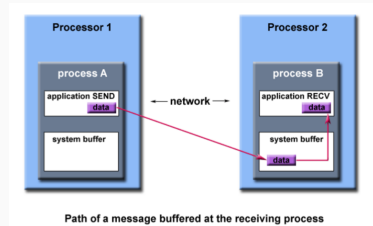
Un mensaje

MPI es un array de elementos de un tipo particular MPI enviado, el cual debe ser el mismo que el recibido. MPI soporta así arquitecturas heterogéneas.

MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Comunicación punto a punto (bloqueada)

Los tipos de operaciones **punto a punto** ejecutan envío de mensajes entre dos procesadores. Envío y recibo seguro hace que la ejecución del proceso se detenga y no retorne hasta que suceda el envío o recibo.



Se clasifican de acuerdo al tipo de envío de mensajes, en:

- **Send sincrónico** , solo se complegta cuando ha sido recibido
- **Buffered Send**, siempre se completa (haya sido recibido o no)
- **Send standard**, sincrónico o buffered
- **Ready Send**, siempre se completa (haya sido recibido o no)
- **Receive**, se completa cuando el mensaje ha sido recibido

Envío y recibo pueden estar sincronizados, durante lo cual se graba la información en memoria (system buffer) reservada para guardar data en tránsito.

MPI_Send (standard):

Operación de envío de mensajes. Tener en cuenta que:

- No se debe asumir que el envío de mensajes se complete antes de que **receive** empiece
- Debe ser sincrónico o buffered



MPI_Send (standard):

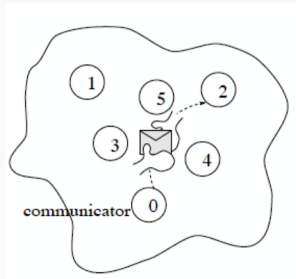
Operación de envío de mensajes.

MPI_Send(&buffer,count,datatype,dest,tag,comm)

- **buffer**: puntero a la variable que ocupa la data que va a ser enviada o recibida (nombre de la variable)
- **count**: número de elementos de la data que seran enviados/recibidos
- **datatype**: tipo de variable MPI
- **dest**: proceso destino del mensaje (rank)
- **tag**: identificador del proceso que envía. Útil en caso de envío de múltiples mensajes
- **comm**: comunicador del grupo de procesos activo

Synchronous Send:

Se utiliza si el proceso que envía debe saber si el mensaje ha sido recibido. Para ello el proceso que recibe manda un mensaje de 'recibido' al enviante, momento en el cual se considera el mensaje como enviado



Synchronous Send:

Operación de envío de mensajes.

```
MPI_SSend(&buffer,count,datatype,dest,tag,comm)
```

- **buffer**: puntero a la variable que ocupa la data que va a ser enviada o recibida (nombre de la variable)
- **count**: número de elementos de la data que serán enviados/recibidos
- **datatype**: tipo de variable MPI
- **dest**: proceso destino del mensaje (rank)
- **tag**: identificador del proceso que envía. Útil en caso de envío de múltiples mensajes
- **comm**: comunicador del grupo de procesos activo

Buffered Send:

Garantiza un término inmediato a través del almacenamiento en un buffer para transmisión posterior si es necesario.

```
MPI_BSend(&buffer,count,datatype,dest,tag,comm)
```

Para ello, el usuario debe destinar suficiente memoria para el programa (en Bytes) utilizando

```
MPI_BUFFER_ATTACH(buffer,size)
```

```
MPI_BUFFER_DETACH(buffer,size)
```

Ready Send:

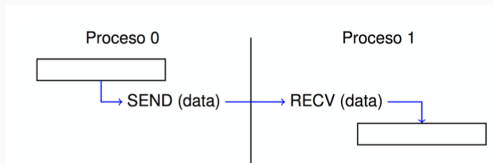
Se completa inmediatamente, enviando el mensaje al comunicador. Este se recibirá solo si el proceso está listo para hacerlo. Si no, ocurrirá un error. El objetivo es mejorar performance. Es un modo difícil para debugging. Solo recomendable si performance es crítica

```
MPI_RSend(&buffer,count,datatype,dest,tag,comm)
```

- **buffer**: puntero a la variable que ocupa la data que va a ser enviada o recibida (nombre de la variable)
- **count**: número de elementos de la data que serán enviados/recibidos
- **datatype**: tipo de variable MPI
- **dest**: proceso destino del mensaje (rank)
- **tag**: identificador del proceso que envía. Útil en caso de envío de múltiples mensajes
- **comm**: comunicador del grupo de procesos activo

MPI_Recv (standard blocking):

Operación de recibo de mensajes. Se completan cuando se confirma por parte del proceso destino



MPI_Recv (standard blocking):

MPI_Recv(&buffer,count,datatype,source,tag,comm,&status)

- **buffer**: puntero a la variable que ocupa la data que va a ser enviada o recibida (nombre de la variable)
- **count**: número de elementos de la data que serán enviados/recibidos
- **datatype**: tipo de variable MPI (el usuario puede crear sus propios tipos de datos)
- **dest**: proceso destino del mensaje (rank)
- **tag**: identificador del mensaje. Útil en caso de envío de múltiples mensajes
- **comm**: comunicador del grupo de procesos activo (MPI_COMM_WORLD)
- **status**: indica el estado del objeto

Estado de la comunicación:

Contiene información del remitente, conocida como *status*

status.MPI_SOURCE contiene información de la fuente del mensaje

status.MPI_TAG contiene información del identificador del mensaje






MPI_GET_COUNT(status,datatype,count) contiene el número de elementos recibidos

Ejemplo 02: Send/Recv (1 a 1)

```
1  int main(int argc, char*argv[])
2  {
3      int rank, size;
4      MPI_Init(&argc,&argv);
5      ....
6      if(rank == 0) {
7          // Si esta activo rank 0, number cambia a -1 y se envia al proceso 1
8          number=-1;
9          MPI_Send(&number,1,MPI_INT,1,0,MPI_COMM_WORLD);
10     }
11     else if(rank == 1) {
12         MPI_Recv(&number,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
13         ...
14     }
15     MPI_Finalize();
16 }
```

Ejemplo 03: Send/Recv (1 a np)

```
1  int main(int argc, char*argv[])
2  {
3      MPI_Init(&argc,&argv);
4      ....
5      if(rank == 0) {
6          buf=1;
7          for(i=1;i<numprocs;i++)
8              MPI_Send(&buf,1,MPI_INT,i,0,MPI_COMM_WORLD);
9      }
10     else{
11         MPI_Recv(&buf,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&stat);
12         ...
13     }
14     MPI_Finalize();
15 }
```

-  David B. Kirk and Wen-mei W. Hwu *Programming Massively Parallel Processors: A Hands-on Approach*. 2nd. Morgan Kaufmann, 2013. isbn: 978-0-12-415992-1.
-  Norm Matloff. *Programming on Parallel Machines*. University of California, Davis, 2014.
-  Peter S. Pacheco. *An Introduction to Parallel Programming*. 1st. Morgan Kaufmann, 2011. isbn: 978-0-12-374260- 5.
-  Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. 1st. McGraw-Hill Education Group, 2003. isbn: 0071232656.
-  Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Program- ming*. 1st. Addison-Wesley Professional, 2010. isbn: 0131387685, 9780131387683.