



# Computación Paralela y Distribuida

2020-II

---

José Fiestas

December 10, 2020

Universidad Nacional de Ingeniería  
[jose.fiestas@uni.edu.pe](mailto: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

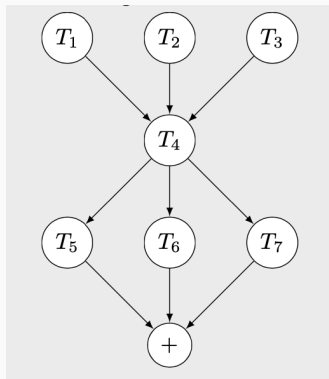
# Comunicación Punto a Punto - DAG

---

# Ejemplo

Identifique  
el grafo de dependencias  
con el código mostrado:

```
1  double calculo(int i,int j,int k)
2  {
3  double a,b,c,d,x,y,z;
4  a = T1(i);
5  b = T2(j);
6  c = T3(k);
7  d = T4(a+b+c);
8  x = T5(a/d);
9  y = T6(b/d);
10 z = T7(c/d);
11 return x+y+z;
12 }
```



¿Por qué  
no se muestran las dependencias  
 $T_1 \rightarrow T_5$ ,  $T_2 \rightarrow T_6$ ,  $T_3 \rightarrow T_7$  ?

## Comunicación no-bloqueada

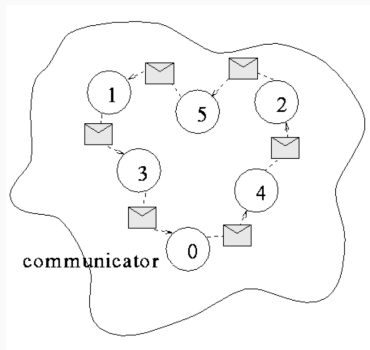
# Comunicación Punto a Punto - bloqueada

Sender mode	Notes	Synchronous?
Synchronous send	Message goes directly to receiver. Only completes when the receive has begun.	<i>synchronous</i>
Buffered send	Message is copied in to a "buffer" (provided by the application). Always completes (unless an error occurs), irrespective of receiver.	<i>asynchronous</i>
Standard send	Either synchronous or buffered (into a fixed size buffer provided by MPI system)	<i>both/hybrid</i>
Ready send	<i>Assumes</i> the matching receive has been called. Undefined behaviour (possibly an error) if the receiver is not ready.	<i>neither?</i>
Receive	Completes when a message has arrived	

Operation	MPI Call
Standard send	MPI_Send
Synchronous send	MPI_Ssend
Buffered send	MPI_Bsend
Ready send	MPI_Rsend
Receive	MPI_Recv

# Comunicación bloqueada

Causa retrasos de procesos a causa de otros que no culminan su trabajo. Deadlock sucede cuando varios procesos esperan para poder continuar su trabajo

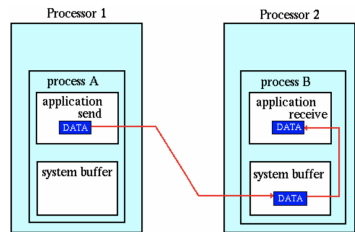


## Buffer (memoria)

Variables o vectores que se usan como argumentos de contenedores de mensajes en rutinas MPI. Hay formas de especificar un espacio de buffer, i.e.

**`MPI_Buffer_attach(void* buffer, int size)`**

El espacio ofrecido por el sistema  
Camino de un mensaje grabado  
(buffered) en el proceso de recibo





Puede iniciarse independientemente del recibo del mensaje, pero se completa solo si se confirma que el mensaje ha sido recibido, para que el buffer pueda volver a ser usado Analogía con el teléfono:

**sincronizado** : se contesta personalmente

**no-sincronizado**: se deja un mensaje

La comunicación no retorna ningún valor hasta que ésta esté terminada (cuando el buffer es libre de nuevo).

En el problema de suavizado sería:

```
for (iterations)
.      update all cells;
.      send boundary values to neighbours;
.      receive halo values from neighbours;
```

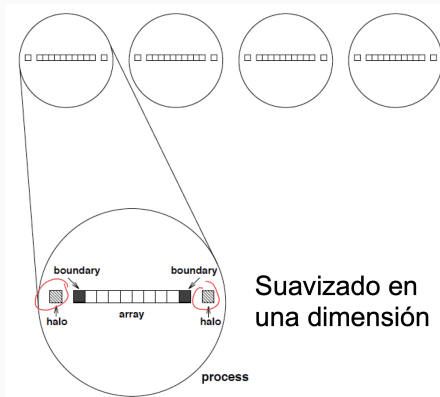
# Comunicación no-bloqueada

Se necesita separar el envío y recibo de información por los vecinos. Para ello, el Send y Recv retornan valores antes de completarse la comunicación, permitiendo hacerse cargo de otros envíos/recibos. Es decir, la comunicación tendrá ahora dos operaciones: de inicio y final.

```
for (iterations)
.   update boundary cells;
.   initiate sending of boundary values to neighbours;
.   initiate receipt of halo values from neighbours;
.   update non-boundary cells;
.   wait for completion of sending of boundary values;
.   wait for completion of receipt of halo values;
```

# Comunicación no bloqueada

Operación de suavizado  
(**smoothing**) promedia  
los vecinos inmediatos.  
Los extremos del sub-array  
necesitan comunicar sus  
valores a procesos vecinos



Ejecuta 3 fases:

- Inicializar la comunicación
- Seguir trabajando (incluso realizando otras comunicaciones)
- Esperar por la ejecución de la comunicación no-bloqueada

**Send no-bloqueado:**

**Analogía:** pedir a la secretaria organize un meeting (se confirma luego y no se sabe si los participantes reciben la notificación)

**Receive no-bloqueado:**

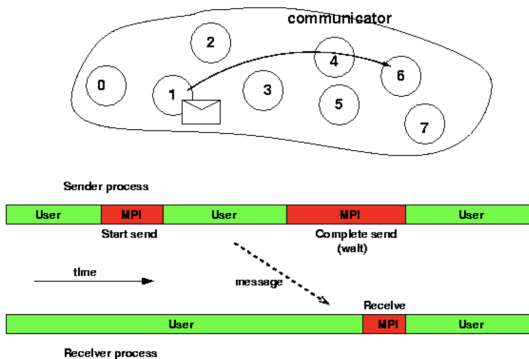
**Analogía:** Esperar recibir regalos en tu cumpleaños, pero no poder pedirlos directamente ...

# Comunicación no-bloqueada

El envío se inicia con **MPI\_Isend**, y continúa con otras comunicaciones

```
int MPI_Isend(void* buf, int count, MPI_Datatype datatype,  
int dest, int tag, MPI_Comm comm, MPI_Request *request)
```

## Non-Blocking Send

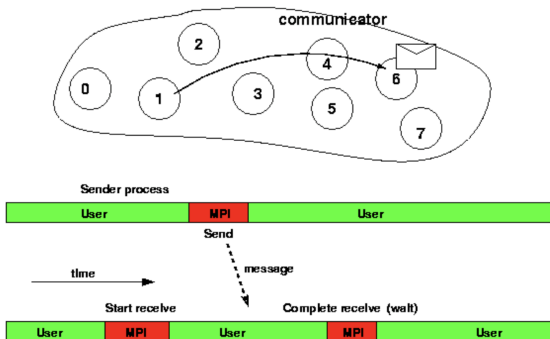


# Comunicación no-bloqueada

El recibo se inicia con **MPI\_Irecv**, y continúa con otras comunicaciones

**int MPI\_Irecv(void\* buf, int count, MPI\_Datatype datatype, int source, int tag, MPI\_Comm comm, MPI\_Request \*request)**

## Non-Blocking Receive



Las operaciones **bloqueadas** solo retornan del procedimiento MPI cuando la operación (envío o recibo) se ha completado.

En caso de operaciones **no bloqueadas**, se retorna inmediatamente antes de ser completado, i.e. el programa ejecuta la siguiente orden, mientras MPI hace la comunicación, y MPI confirma el fin de la operación mas tarde.

La comunicación no-bloqueada cuenta con **operaciones de espera** (matching wait), para que la memoria no se libere hasta que **wait** ha sido llamado



Es necesario saber si la comunicación terminó antes de utilizar el resultado del proceso o de re- utilizar el buffer. Hay dos tipos de control:

1. **Wait type**, que bloquean la comunicación hasta que culmine. Es equivalente a comunicación bloqueada
2. **Test type**, que retorna TRUE o FALSE dependiendo si la comunicación ha culminado o no, pero no bloquean.

```
int MPI_Wait(MPI_Request *req, MPI_Status *status)
```

Se declara luego de **ISend** y **Irecv**, los cuales detienen su operación hasta que se completa la operación referenciada por Request **\*req**. (manejo de la operacion - handle) El estado de la operación esta referenciado por **\*status**

```
int MPI_Test(MPI_Request *req, int *flag, MPI_Status *status)
```

Solo observa el fin de determinada operación. Flag es un puntero a entero que contendrá el resultado del test (true o false)

MPI prueba el término de comunicación múltiple controlando todas (**all**) las comunicaciones, cualquiera (**any**) de ellas, o algunas (**some**) de ellas

Test for completion	WAIT type (blocking)	TEST type (query only)
At least one, return exactly one	MPI_WAITANY	MPI_TESTANY
Every one	MPI_WAITALL	MPI_TESTALL
At least one, return all which completed	MPI_WAITSOME	MPI_TESTSOME

# Término de todas las comunicaciones

```
int MPI_Waitall(int count, MPI_Request *req, MPI_Status *status);
```

Espera el final de todas las operaciones referenciadas por **\*req**  
**count** es el tamaño de la lista de req (y status), normalmente el numero de procesos

```
int MPI_Testall (int count, MPI_Request *req, flag, MPI_Status *status);
```

Si todos terminan, flag is TRUE

# Término de cualquiera de las comunicaciones

```
int MPI_Waitany(int count, MPI_Request *req, index,  
MPI_Status *status);
```

Espera el final de alguna de las operaciones referenciadas por **\*req**  
**count** es el tamaño de la lista de req (y status), index da la posición del  
proceso terminado. **count** es el tamaño de la lista de req (y status),  
normalmente el numero de procesos

```
MPI_Testany (int count, MPI_Request *req, index, flag,  
MPI_Status *status);
```

flag (TRUE/FALSE) contiene el resultado del test

```
int MPI_Waitsome(int count, MPI_Request *req, int out-  
count, *index, MPI_Status *status);
```

Similar a MPI\_Waitany, MPI\_Testany, pero retorna el status de todas las comunicaciones terminadas

```
MPI_Testsome (int count, MPI_Request *req,int outcount,  
*index, MPI_Status *status);
```

Criterios para programar correcta y eficientemente en paralelo:

## **Sincronizado vs. no-sincronizado:**

Describe el tiempo relativo a mensajes enviados o recibidos.

## **Bloqueado vs. No-bloqueado:**

Describe fin de la operación en el que envía o recibe, independientemente



**Ejemplo:** proceso sincronizado y no-bloqueado

```
MPI_Request request;  
MPI_Isend(buf,count,datatype,  
dest,tag,comm,&request);  
MPI_Wait(&request,&status);
```

```
MPI_Request request;  
MPI_Irecv(buf,count,datatype,  
src,tag,comm,&request);  
MPI_Wait(&request,&status);
```

- Send y Recv pueden ser bloqueados o no-bloqueados
- Un Send bloqueado puede ser usado con un Recv no- bloqueado, y viceversa
- Send no-bloqueados pueden usar cualquier modo (sincronizado, buffered, standard, o ready)

## Communication Modes

Non-Blocking Operation	MPI Call
Standard send	MPI_Isend
Synchronous send	MPI_Issend
Buffered Send	MPI_IbSEND
Ready send	MPI_IrSEND
Receive	MPI_Irecv

## Ejercicios

---

# Ejercicio 1: Promedio

Programar el cálculo del promedio de un array de floats aleatorios, en el rango de  $[0, 1000]$ , utilizando comunicación no-bloqueada (`MPI_Isend`, `MPI_Irecv`). Generar un array de  $N=10^6$  elementos, con el objetivo de medir tiempos significativos de proceso, y comparar con los tiempos utilizando comunicación bloqueada (`MPI_Send`, `MPI_Recv`)






Utilice  $p=1,2,4,8$  y comente los resultados comparando bloqueo y no-bloqueo

Nota: repitan el calculo varias veces para que sea estadisticamente correcto

## Ejercicio 2

Implemente un DAG y un código del ejemplo de clase utilizando operaciones de comunicación colectiva.

¿Permite esto reducir las operaciones de comunicación utilizadas en el ejemplo?

-  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.