

Programación Concurrente ATIC

Redictado de Programación Concurrente

Clase 11



Facultad de Informática
UNLP

Links al archivo con audio

La teoría con los audios está en formato MP4. Debe descargar los archivos comprimidos de los siguientes links:

- ◆ Paradigmas de Interacción entre Procesos:

<https://drive.google.com/uc?id=1a0QUfXjpdM26pTIGzSEjm7zGRtCxAs9d&export=download>

- ◆ Librería para Pasaje de Mensajes (MPI):

<https://drive.google.com/uc?id=1tlOM5BaPD2KSIRIUVYWnwO-8SDqLPIE8&export=download>



Paradigmas de Interacción entre Procesos

Paradigmas para la interacción entre procesos

- 3 esquemas básicos de interacción entre procesos: *productor/consumidor*, *cliente/servidor* e *interacción entre pares*.
- Estos esquemas básicos se pueden combinar de muchas maneras, dando lugar a otros **paradigmas** o modelos de interacción entre procesos.

Paradigma 1: *master / worker*

Implementación distribuida del modelo *Bag of Task*.

Paradigma 2: *algoritmos heartbeat*

Los procesos periódicamente deben intercambiar información con mecanismos tipo send/receive.

Paradigma 3: *algoritmos pipeline*

La información recorre una serie de procesos utilizando alguna forma de receive/send.

Paradigmas para la interacción entre procesos

Paradigma 4: *probes (send) y echoes(receive)*

La interacción entre los procesos permite recorrer grafos o árboles (o estructuras dinámicas) disseminando y juntando información.

Paradigma 5: *algoritmos broadcast*

Permiten alcanzar una información global en una arquitectura distribuida. Sirven para toma de decisiones descentralizadas.

Paradigma 6: *token passing*

En muchos casos la arquitectura distribuida recibe una información global a través del viaje de tokens de control o datos. También permite la toma de decisiones distribuidas.

Paradigma 7: *servidores replicados*

Los servidores manejan (mediante múltiples instancias) recursos compartidos tales como dispositivos o archivos.

Paradigmas para la interacción entre procesos

Manager/Worker

- El concepto de *bag of tasks* usando variables compartidas supone que un conjunto de workers comparten una “bolsa” con tareas independientes. Los workers sacan una tarea de la bolsa, la ejecutan, y posiblemente crean nuevas tareas que ponen en la bolsa (ejemplo en LINDA manejando un espacio compartido de tuplas).
- La mayor virtud de este enfoque es la escalabilidad y la facilidad para equilibrar la carga de trabajo de los workers.
- Analizaremos la implementación de este paradigma con mensajes en lugar de MC. Para esto un proceso *manager* implementará la “bolsa” manejando las tasks, comunicándose con los workers y detectando fin de tareas. **Se trata de un esquema C/S.**
- Ejemplo: multiplicación de matrices ralas.

Paradigmas para la interacción entre procesos

Heartbeat

- Paradigma *heartbeat* \Rightarrow útil para soluciones iterativas que se quieren paralelizar.
- Usando un esquema “*divide & conquer*” se distribuye la carga (datos) entre los workers; cada uno es responsable de actualizar una parte.
- Los nuevos valores dependen de los mantenidos por los workers o sus vecinos inmediatos.
- Cada “paso” debiera significar un progreso hacia la solución.
- Formato general de los worker:

```
process worker [i = 1 to numWorkers]
{  declaraciones e inicializaciones locales;
  while (no terminado)
  {  send valores a los workers vecinos;
    receive valores de los workers vecinos;
    Actualizar valores locales;
  }
}
```

- Ejemplo: grid computations (imágenes), autómatas celulares (simulación de fenómenos como incendios o crecimiento biológico).

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

Los procesadores están conectados por canales bidireccionales. Cada uno se comunica sólo con sus vecinos y conoce esos links.

¿Cómo puede cada procesador determinar la topología completa de la red?

➤ Modelización:

- Procesador \Rightarrow proceso
- Links de comunicación \Rightarrow canales compartidos.

➤ Soluciones: los vecinos interactúan para intercambiar información local.

Algoritmo Heartbeat: se expande enviando información; luego se contrae incorporando nueva información.

➤ Procesos *Nodo*[$p:1..n$].

➤ Vecinos de p : *vecinos*[$1:n$] \rightarrow *vecinos*[q] es true si q es vecino de p .

➤ **Problema:** computar *top* (matriz de adyacencia), donde *top*[p,q] es true si p y q son vecinos.

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

Cada nodo debe ejecutar un n° de rondas para conocer la topología completa. Si el diámetro D de la red es conocido se resuelve con el siguiente algoritmo.

```
chan topologia[1:n] ([1:n,1:n] bool)

Process Nodo[p:1..n]
{ bool vecinos[1:n], bool nuevatop[1:n,1:n], top[1:n,1:n] = ([n*n] false);
  top[p,1..n] = vecinos;

  for (r = 0 ; r < D; r++)
    { for [q = 1 to n st vecinos[q] ] send topologia[q](top);
      for [q = 1 to n st vecinos[q] ]
        { receive topologia[p](nuevatop);
          top = top or nuevatop;
        }
    }
}
```

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

- Rara vez se conoce el valor de D .
- Excesivo intercambio de mensajes \Rightarrow los procesos cercanos al “centro” conocen la topología más pronto y no aprenden nada nuevo en los intercambios.
- El tema de la terminación \Rightarrow ¿local o distribuida?
- *¿Cómo se pueden solucionar estos problemas?*
 - Después de r rondas, p conoce la topología a distancia r de él. Para cada nodo q dentro de la distancia r de p , los vecinos de q estarán almacenados en la fila q de $top \Rightarrow p$ ejecutó las rondas suficientes tan pronto como cada fila de top tiene algún valor *true*.
 - Luego necesita ejecutar una última ronda para intercambiar la topología con sus vecinos.
- No siempre la terminación se puede determinar localmente.

Paradigmas para la interacción entre procesos

Heartbeat - Topología de una red

```
chan topologia[1:n](emisor : int; listo : bool; top : [1:n,1:n] bool)
```

```
Process Nodo[p:1..n]
```

```
{ bool vecinos[1:n], activo[1:n] = vecinos, top[1:n,1:n] = ([n*n]false), nuevatop[1:n,1:n];  
  bool qlisto, listo = false;  
  int emisor;  
  top[p,1..n] = vecinos;  
  while (not listo)  
  {  
    for [q = 1 to n st activo[q] ] send topologia[q](p,false,top);  
    for [q = 1 to n st activo[q] ]  
    {  
      receive topologia[p](emisor,qlisto,nuevatop);  
      top = top or nuevatop;  
      if (qlisto) activo[emisor] = false;  
    }  
    if (todas las filas de top tiene 1 entry true) listo=true;  
  }  
  for [q = 1 to n st activo[q] ] send topologia[q](p,listo,top);  
  for [q=1 to n st activo[q]] receive topologia[p](emisor,d,nuevatop);  
}
```

Paradigmas para la interacción entre procesos

Pipeline

- Un pipeline es un arreglo lineal de procesos “filtro” que reciben datos de un puerto (canal) de entrada y entregan resultados por un canal de salida.
- Estos procesos (“workers”) pueden estar en procesadores que operan en paralelo, en un primer esquema *a lazo abierto* (W_1 en el INPUT, W_n en el OUTPUT).
- Un segundo esquema es el pipeline *circular*, donde W_n se conecta con W_1 . Estos esquemas sirven en procesos iterativos o bien donde la aplicación no se resuelve en una pasada por el pipe.
- En un tercer esquema posible (*cerrado*), existe un proceso coordinador que maneja la “realimentación” entre W_n y W_1 .
- Ejemplo: multiplicación de matrices en bloques.

Paradigmas para la interacción entre procesos

Probe-Echo

- Árboles y grafos son utilizados en muchas aplicaciones distribuidas como búsquedas en la WEB, BD, sistemas expertos y juegos.
- Las arquitecturas distribuidas se pueden asimilar a los nodos de grafos y árboles, con canales de comunicación que los vinculan.
- DFS es uno de los paradigmas secuenciales clásicos para visitar todos los nodos en un árbol o grafo. Este paradigma es el análogo concurrente de DFS.
- **Prueba-eco** se basa en el envío de un mensajes (“probe”) de un nodo al sucesor, y la espera posterior del mensaje de respuesta (“echo”).
- Los **probes** se envían en paralelo a todos los sucesores.
- Los algoritmos de prueba-eco son particularmente interesantes cuando se trata de recorrer redes donde no hay (o no se conoce) un número fijo de nodos activos (ejemplo: redes móviles).

Paradigmas para la interacción entre procesos

Broadcast

- En la mayoría de las LAN cada procesador se conecta directamente con los otros. Estas redes normalmente soportan la primitiva ***broadcast***:

broadcast ch(m);

- Los mensajes broadcast de un proceso se encolan en los canales en el orden de envío, pero broadcast no es atómico y los mensajes enviados por procesos A y B podrían ser recibidos por otros en distinto orden.
- Se puede usar broadcast para diseminar información o para resolver problemas de sincronización distribuida. Ejemplo: semáforos distribuidos, la base es un ***ordenamiento total de eventos de comunicación*** mediante el uso de ***relojes lógicos***.

Paradigmas para la interacción entre procesos

Token Passing

- Un paradigma de interacción muy usado se basa en un tipo especial de mensaje (“token”) que puede usarse para otorgar un permiso (control) o recoger información global de la arquitectura distribuida. Un ejemplo del primer tipo de algoritmos es el caso de tener que controlar *exclusión mutua distribuida*.
- Ejemplos de recolección de información de estado son los algoritmos de detección de terminación en computación distribuida.
- Aunque el problema de la SC se da principalmente en programas de MC, puede encontrarse en programas distribuidos cuando hay algún recurso compartido que puede usar un único proceso a la vez. Generalmente es una componente de un problema más grande, tal como asegurar consistencia en un sistema de BD.
- Soluciones posibles: Monitor activo que da permiso de acceso (ej: locks en archivos), semáforos distribuidos (usando broadcast, con gran intercambio de mensajes), o *token ring* (descentralizado y fair).

Paradigmas para la interacción entre procesos

Servidores Replicados

- Un server puede ser replicado cuando hay múltiples instancias de un recurso: cada server maneja una instancia.
- La replicación también puede usarse para darle a los clientes la sensación de un único recurso cuando en realidad hay varios.
- Ejemplo: problema de los filósofos
 - Modelo **centralizado**: los Filósofo se comunican con **UN** proceso Mozo que decide el acceso o no a los recursos.
 - Modelo **distribuido**: supone **5 procesos Mozo**, cada uno manejando un tenedor. Un Filósofo puede comunicarse con **2** Mozos (izquierdo y derecho), solicitando y devolviendo el recurso. Los Mozos **NO se comunican entre ellos**.
 - Modelo **descentralizada**: cada Filósofo ve **un único** Mozo. Los Mozos se comunican entre ellos (cada uno con sus **2** vecinos) para decidir el manejo del recurso asociado a “su” Filósofo.



Librerías para manejo de PM

Operaciones Send y Receive

- Los prototipos de las operaciones son:

Send (void *sendbuf, int nelems, int dest)

Receive (void *recvbuf, int nelems, int source)

- Ejemplo:

P0

```
a = 100;  
send(&a, 1, 1);  
a = 0;
```

P1

```
receive(&a, 1, 0)  
printf("%d\n", a);
```

- La semántica del SEND requiere que en P1 quede el valor 100 (no 0).
- Diferentes protocolos para Send y Receive.

Send y Receive bloqueante

- Para asegurar la semántica del SEND → no devolver el control del Send hasta que el dato a transmitir esté seguro (Send bloqueante).
- Ociosidad del proceso.
- Hay dos posibilidades:
 - Send/Receive bloqueantes sin buffering.
 - Send/Receive bloqueantes con buffering.

Send y Receive no bloqueante

- Para evitar overhead (ociosidad o manejo de buffer) se devuelve el control de la operación inmediatamente.
- Requiere un posterior chequeo para asegurarse la finalización de la comunicación.
- Deja en manos del programador asegurar la semántica del SEND.
- Hay dos posibilidades:
 - Send/Receive no bloqueantes sin buffering.
 - Send/Receive no bloqueantes con buffering.



MPI

Message Passing Interface

Librería MPI (Interfaz de Pasaje de Mensajes)

- Existen numerosas librerías para pasaje de mensaje (no compatibles).
- MPI define una librería estándar que puede ser empleada desde C o Fortran (y potencialmente desde otros lenguajes).
- El estándar MPI define la sintaxis y la semántica de más de 125 rutinas.
- Hay implementaciones de MPI de la mayoría de los proveedores de hardware.
- Modelo SPMD.
- Todas las rutinas, tipos de datos y constantes en MPI tienen el prefijo “MPI_”. El código de retorno para operaciones terminadas exitosamente es MPI_SUCCESS.
- Básicamente con 6 rutinas podemos escribir programas paralelos basados en pasaje de mensajes: MPI_Init, MPI_Finalize, MPI_Comm_size, MPI_Comm_rank, MPI_Send y MPI_Recv.

Librería MPI - Inicio y finalización de MPI

- ***MPI_Init***: se invoca en todos los procesos antes que cualquier otro llamado a rutinas MPI. Sirve para inicializar el entorno MPI.

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

Algunas implementaciones de MPI requieren argc y argv para inicializar el entorno

- ***MPI_Finalize***: se invoca en todos los procesos como último llamado a rutinas MPI. Sirve para cerrar el entorno MPI.

`MPI_Finalize ()`

Librería MPI - Comunicadores

- Un comunicador define el dominio de comunicación.
- Cada proceso puede pertenecer a muchos comunicadores.
- Existe un comunicador que incluye a todos los procesos de la aplicación `MPI_COMM_WORLD`.
- Son variables del tipo `MPI_Comm` → almacena información sobre que procesos pertenecen a él.
- En cada operación de transferencia se debe indicar el comunicador sobre el que se va a realizar.

Librería MPI - Adquisición de Información

- ***MPI_Comm_size***: indica la cantidad de procesos en el comunicador.

`MPI_Comm_size (MPI_Comm comunicador, int *cantidad).`

- ***MPI_Comm_rank***: indica el “rank” (identificador) del proceso dentro de ese comunicador.

`MPI_Comm_rank (MPI_Comm comunicador, int *rank)`

- rank es un valor entre [0..cantidad]
- Cada proceso puede tener un rank diferente en cada comunicador.

EJEMPLO: `#include <mpi.h>`

```
main(int argc, char *argv[])
{
    int cantidad, identificador;

    MPI_Init(&argc, &argv);
    MPI_Comm_size (MPI_COMM_WORLD, &cantidad);
    MPI_Comm_rank(MPI_COMM_WORLD, &identificador);
    printf("Soy %d de %d \n", identificador, cantidad);
    MPI_Finalize();
}
```

Librería MPI - Tipos de Datos para las comunicaciones

Tipo de Datos MPI	Tipo de Datos C
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	

Librería MPI - Comunicación punto a punto

➤ Diferentes protocolos para Send.

- Send bloqueantes con buffering (Bsend).
- Send bloqueantes sin buffering (Ssend).
- Send no bloqueantes (Isend).

➤ Diferentes protocolos para Recv.

- Recv bloqueantes (Recv).
- Recv no bloqueantes (Irecv).

Librería MPI - Comunicación bloqueante punto a punto

- `MPI_Send`, `MPI_Ssend`, `MPI_Bsend`: rutina básica para enviar datos a otro proceso.

`MPI_Send` (`void *buf`, `int cantidad`, `MPI_Datatype tipoDato`, `int destino`, `int tag`, `MPI_Comm comunicador`)

- Valor de Tag entre `[0..MPI_TAG_UB]`.

- `MPI_Recv`: rutina básica para recibir datos a otro proceso.

`MPI_Recv` (`void *buf`, `int cantidad`, `MPI_Datatype tipoDato`, `int origen`, `int tag`, `MPI_Comm comunicador`, `MPI_Status *estado`)

- Comodines `MPI_ANY_SOURCE` y `MPI_ANY_TAG`.
- Estructura `MPI_Status`

```
typedef struct MPI_Status { int MPI_SOURCE;  
                           int MPI_TAG;  
                           int MPI_ERROR; }
```

- `MPI_Get_count` para obtener la cantidad de elementos recibidos
`MPI_Get_count(MPI_Status *estado, MPI_Datatype tipoDato, int *cantidad)`

Ejemplo

Dos procesos intercambian valores (14 y 25). Solución empleando MPI:

```
#include <mpi.h>
main (INT argc, CHAR *argv [ ]) {
    INT id, idAux;
    INT longitud=1;
    INT valor, otroValor;
    MPI_status estado;

    MPI_Init (&argc, &argv);

    MPI_Comm_Rank (MPI_COMM_WORLD, &id);
    IF (id == 0) { idAux = 1; valor = 14;}
    ELSE { idAux = 0; valor = 25; }

    MPI_send (&valor, longitud, MPI_INT, idAux, 1, MPI_COMM_WORLD);
    MPI_recv (&otroValor, 1, MPI_INT, idAux, 1, MPI_COMM_WORLD, &estado);
    printf ("process %d received a %d\n", id, otroValor);
    MPI_Finalize ( );
}
```

Ejemplo

En este caso resolvemos el mismo ejercicio pero para que no haya Deadlock si el Send actúa como Ssend.

```
#include <mpi.h>
main (INT argc, CHAR *argv [ ]) {
    INT id;
    INT valor, otroValor;
    MPI_status estado;

    MPI_Init (&argc, &argv);
    MPI_Comm_Rank (MPI_COMM_WORLD, &id);
    IF (id == 0) { valor = 14;
        MPI_send (&valor, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
        MPI_recv (&otroValor, 1, MPI_INT, 1, 1, MPI_COMM_WORLD, &estado);
    }
    ELSE { valor = 25;
        MPI_recv (&otroValor, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &estado);
        MPI_send (&valor, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }
    printf ("process %d received a %d\n", id, otroValor);
    MPI_Finalize ();
}
```

Librería MPI - Comunicación no bloqueante punto a punto

- Comienzan la operación de comunicación e inmediatamente devuelven el control (no se asegura que la comunicación finalice correctamente).

`MPI_Isend (void *buf, int cantidad, MPI_Datatype tipoDato, int destino, int tag, MPI_Comm comunicador, MPI_Request *solicitud)`

`MPI_Irecv (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, int tag, MPI_Comm comunicador, MPI_Request *solicitud)`

- `MPI_Test`: testea si la operación de comunicación finalizó.

`MPI_Test (MPI_Request *solicitud, int *flag, MPI_Status *estado)`

- `MPI_Wait`: bloquea al proceso hasta que finaliza la operación.

`MPI_Wait (MPI_Request *solicitud, MPI_Status *estado)`

- Este tipo de comunicación permite solapar computo con comunicación. Evita overhead de manejo de buffer. Deja en manos del programador asegurar que se realice la comunicación correctamente.

Librería MPI - Comunicación no bloqueante punto a punto

Código usando comunicación bloqueante

```
EJEMPLO: main (int argc, char *argv[])
{
    int cant, id, *dato, i;
    MPI_Status estado;

    dato = (int *) malloc (100 * sizeof(int));
    MPI_Init(&argc,&argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);

    if (id == 0)
    {
        cant = atoi(argv[1])%100;
        MPI_Send(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD);
        for (i=0; i< 100; i++) dato[i]=0;
    }
    else
    {
        MPI_Recv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD, &estado);
        MPI_Get_count(&estado, MPI_INT, &cant);
        //PROCESA LOS DATOS;
    };
    MPI_Finalize;
}
```

Para usar comunicación NO bloqueante (¿alcanza con cambiar el Send por Isend?)

Librería MPI - Comunicación no bloqueante punto a punto

Código anterior usando comunicación no bloqueante

```
EJEMPLO: main (int argc, char *argv[])
{
    int cant, id, *dato, i;
    MPI_Status estado;
    MPI_Request req;

    dato = (int *) malloc (100 * sizeof(int));
    MPI_Init(&argc,&argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);

    if (id == 0)
    {
        cant = atoi(argv[1]);
        //INICIALIZA dato
        MPI_Isend(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD, &req);
        //TRABAJA
        MPI_Wait(&req, &estado);
        for (i=0; i< 100; i++) dato[i]=0;
    }
    else
    {
        MPI_Recv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD, &estado);
        MPI_Get_count(&estado, MPI_INT, &cant);
        //PROCESA LOS DATOS;
    };
    MPI_Finalize;
}
```

Librería MPI - Comunicación no bloqueante punto a punto

```
EJEMPLO: main (int argc, char *argv[])
{
    int id, *dato, i, flag;
    MPI_Status estado;
    MPI_Request req;

    dato = (int *) malloc (100 * sizeof(int));
    MPI_Init(&argc,&argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);

    if (id == 0)
    { //INICIALIZA dato
        MPI_Send(dato,cant,MPI_INT,1,1,MPI_COMM_WORLD);
    }
    else
    { MPI_Irecv(dato,100,MPI_INT,0,1,MPI_COMM_WORLD ,&req);
      MPI_Test(&req, &flag,&estado);
      while (!flag)
      { //Trabaja mientras espera
        MPI_Test(&req, &flag,&estado);
      };
      //PROCESA LOS DATOS;
    };
    MPI_Finalize;
}
```

Librería MPI – Consulta de mensajes pendientes

- Información de un mensaje antes de hacer el Recv (Origen, Cantidad de elementos, Tag).
- MPI_Probe: bloquea el proceso hasta que llegue un mensaje que cumpla con el origen y el tag.

MPI_Probe (int origen, int tag, MPI_Comm comunicador, MPI_Status *estado)

- MPI_Iprobe: chequea por el arribo de un mensaje que cumpla con el origen y tag.

MPI_Iprobe (int origen, int tag, MPI_Comm comunicador, int *flag, MPI_Status *estado)

- Comodines en Origen y Tag.

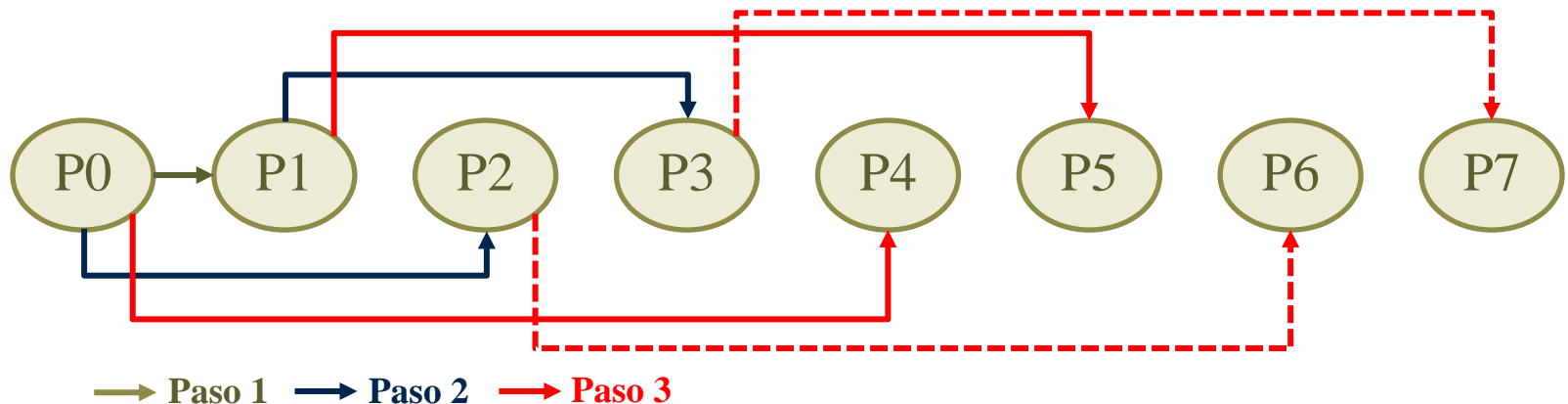
¿Cuándo y porque usar cada uno?

Librería MPI - Comunicaciones Colectivas

MPI provee un conjunto de funciones para realizar operaciones colectivas, sobre un grupo de procesos asociado con un comunicador. Todos los procesos del comunicador deben llamar a la rutina colectiva:

- MPI_Barrier
- MPI_Bcast
- MPI_Scatter - MPI_Scatterv
- MPI_Gather - MPI_Gatherv
- MPI_Reduce
- Otras...

Ventajas del uso de comunicaciones colectivas.



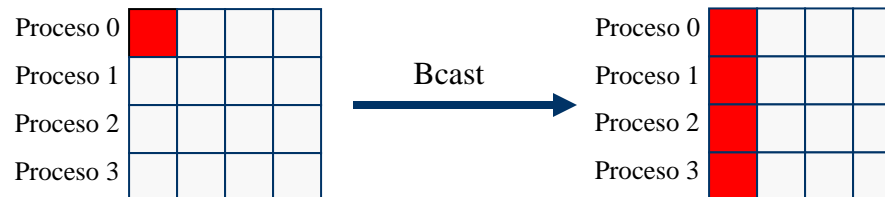
Librería MPI - Comunicaciones Colectivas

- Sincronización en una barrera.

`MPI_Barrier(MPI_Comm comunicador)`

- Broadcast: un proceso envía el mismo mensaje a todos los otros procesos (incluso a él) del comunicador.

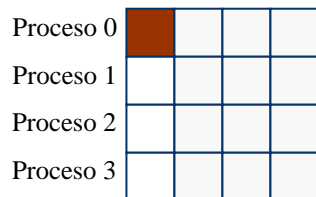
`MPI_Bcast (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, MPI_Comm comunicador)`



Librería MPI - Comunicaciones Colectivas (cont.)

- Reducción de todos a uno: combina los elementos enviados por cada uno de los procesos (inclusive el destino) aplicando una cierta operación.

MPI_Reduce (void *sendbuf, void *recvbuf, int cantidad, MPI_Datatype tipoDato, MPI_Op operación, int destino , MPI_Comm comunicador)



Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

Librería MPI - Comunicaciones Colectivas (cont.)

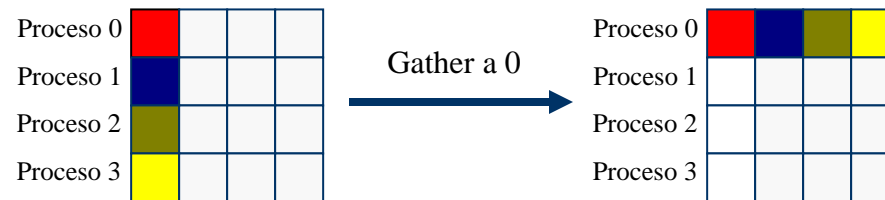
- Gather: recolecta el vector de datos de todos los procesos (inclusive el destino) y los concatena en orden para dejar el resultado en un único proceso.

- Todos los vectores tienen igual tamaño.

`MPI_Gather` (void *sendbuf, int cantEnvio, MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec, MPI_Datatype tipoDatoRec, int destino, MPI_Comm comunicador)

- Los vectores pueden tener diferente tamaño.

`MPI_Gatherv` (void *sendbuf, int cantEnvio, MPI_Datatype tipoDatoEnvio, void*recvbuf, int *cantsRec, int *desplazamientos, MPI_Datatype tipoDatoRec, int destino, MPI_Comm comunicador)



Librería MPI - Comunicaciones Colectivas (cont.)

- Scatter: reparte un vector de datos entre todos los procesos (inclusive el mismo dueño del vector).

- Reparte en forma equitativa (a todos la misma cantidad).

MPI_Scatter (void *sendbuf, int cantEnvio, MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec, MPI_Datatype tipoDatoRec, int origen, MPI_Comm comunicador)

- Puede darle a cada proceso diferente cantidad de elementos.

MPI_Scatterv (void *sendbuf, int *cantsEnvio, int *desplazamientos, MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec, MPI_Datatype tipoDatoRec, int origen, MPI_Comm comunicador)



Minimizando los overheads de comunicación.

- Maximizar la localidad de datos.
- Minimizar el volumen de intercambio de datos.
- Minimizar la cantidad de comunicaciones.
- Considerar el costo de cada bloque de datos intercambiado.
- Replicar datos cuando sea conveniente.
- Lograr el overlapping de cómputo (procesamiento) y comunicaciones.
- En lo posible usar comunicaciones asincrónicas.
- Usar comunicaciones colectivas en lugar de punto a punto