Concurrencia y Paralelismo

Clase 8

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:

- Librería para Pasaje de Mensajes (MPI): https://drive.google.com/uc?id=1tlOM5BaPD2KSlRIUVYWnwO-8SDqLPlE8&export=download

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 trasmitir 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_OMM_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: cheqea 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.

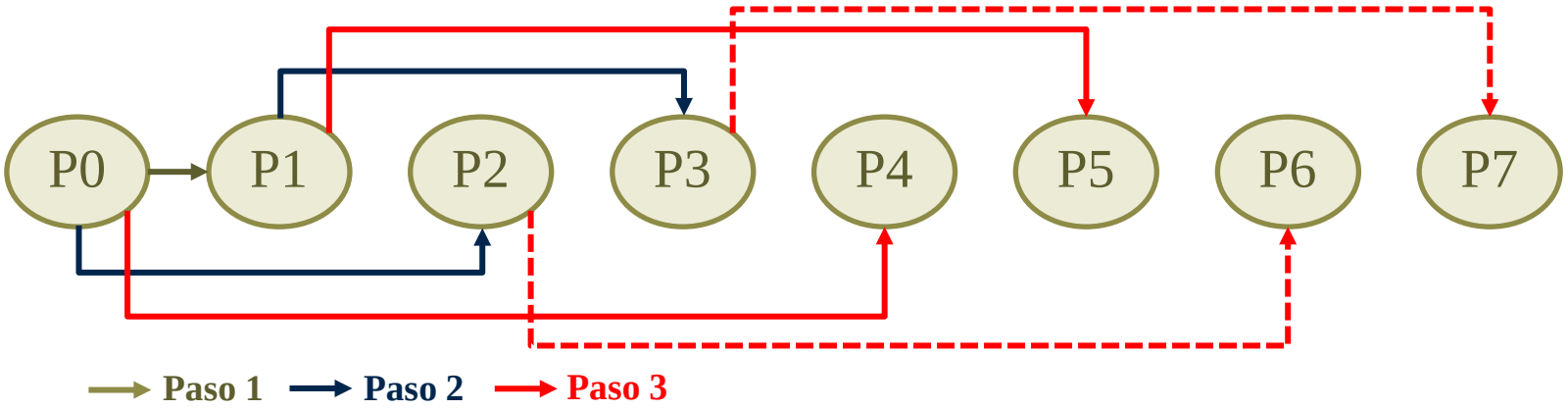
¿Cuando y porque usar cada uno?

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.



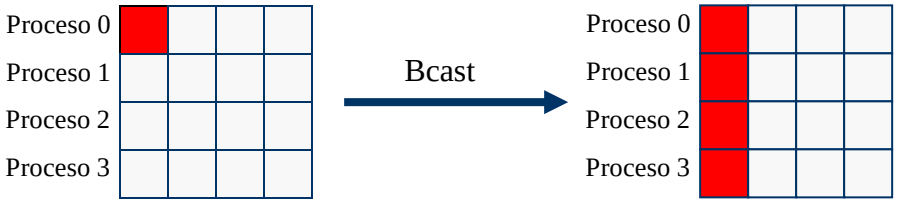
Librería MPI - Comunicaciones Colectivas

- 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.

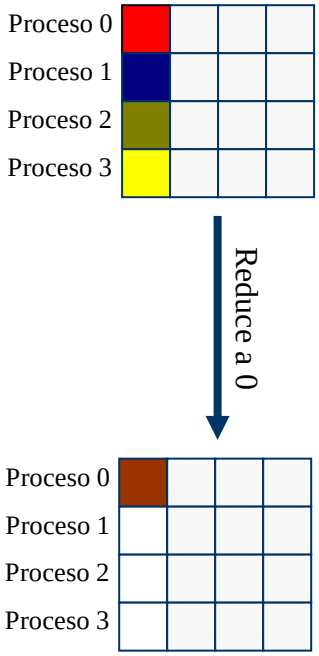
MPI_Bcast (void *buf, int cantidad, MPI_Datatype tipoDato, int origen, MPI_Comm comunicador)



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

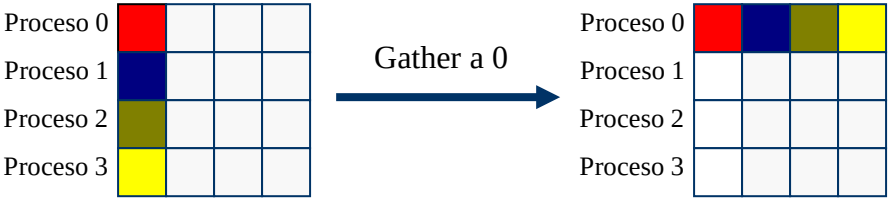
Librería MPI - Comunicaciones Colectivas (cont.)

- 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)



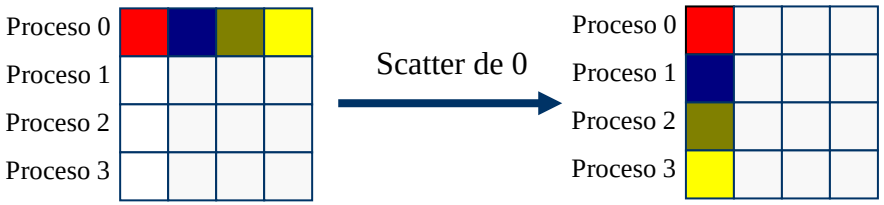
Librería MPI - Comunicaciones Colectivas (cont.)

- 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