Modelos programación HPC: MPI

Paralelismo y Sistemas Distribuidos

Grado en ciencia e ingeniería de datos

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC)

Licencia

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es

Índice

- MPI conceptos básicos
- Rangos MPI
- Envío/ Recepción de mensajes: Comunicación punto a punto
- Envio /Recepción de mensajes: Síncronos vs asíncronos
- Sincronizaciones
- Envio/Recepción de mensajes: Collectivas

Conceptos básicos

- MPI se basa en un modelo de paso de mensajes, donde cada elemento de computación es un proceso y la comunicación entre ellos es explícita
- Los procesos forman parte de un grupo, un comunicador, inicialmente se crea uno que incluye a todos : MPI_COMM_WORD
- MPI permite crear comunicadores nuevos para hacer comunicaciones entre procesos concretos fácilmente
  - Puede ser un duplicado de un comunicador ya existente
  - Puede ser un subgrupo
- Los procesos tienen asignado un identificador (Rank) por comunicador, que va de 0..tamaño_comunicador-1
- Los mensajes deben incluir el tipo de datos, número de elementos, etc
  - MPI define tipos básicos pero soporta definir tipos nuevos específicos del usuario

Hello word

- Debemos incluir mpi.h
- Creacion: MPI_Init(int *argc , char ***argv)
- Finalización:MPI_Finalize()
- Identificación: MPI_Comm_size(MPI_Comm comm , int *size), MPI_comm_rank(MPI_Comm comm , int *rank)

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
    int my_rank, my_size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &my_size);
    printf(”Hi, I’m Rank %d, total processes= %d\n", my_rank, my_size);
    MPI_Finalize();
    return 0;
}
```

Enviar/Recibir

- El envío/recepción puede ser entre procesos concretos (comunicación punto a punto) o entre grupos (colectivas). También encontramos la versión síncrona o asíncrona.
- Síncrona: El proceso espera hasta que los datos son enviados o recibidos
- Asíncrona: El proceso inicia la operación pero retorna inmediatamente, ofrece un *handler* que nos permite consultar el estado de la operación y sincronizarnos (esperar a que termine)
- Hay que especificar el tipo de datos

Tipos datos MPI para C (hay mas)

- MPI_CHAR char
- (treated as printable character)
- MPI_SHORT signed short int
- **MPI_INT signed int**
- MPI_LONG signed long int
- MPI_LONG_LONG_INT signed long long int
- MPI_LONG_LONG (as a synonym) signed long long int
- MPI_SIGNED_CHAR signed char
- (treated as integral value)
- MPI_UNSIGNED_CHAR unsigned char
- (treated as integral value)
- MPI_UNSIGNED_SHORT unsigned short int
- **MPI_UNSIGNED unsigned int**
- MPI_UNSIGNED_LONG unsigned long int
- MPI_UNSIGNED_LONG_LONG unsigned long long int
- MPI_FLOAT float
- **MPI_DOUBLE double**
- MPI_LONG_DOUBLE long double

https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

Enviar (síncrono)

- Operaciones síncronas punto a punto (entre dos procesos).
- Puede bloquear al thread que inicia la llamada (es thread-safe)
- int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

  - buf - dirección inicial del buffer para enviar
  - count - número de elementos en el buffer
  - dtyp - tipo de cada elemento en el buffer
  - dest - rango mpi del destino (0..Rank_size-1)
  - tag - etiqueta para identificar el mensaje
  - comm - comunicador

  inputs

- Acepta MPI_ANY_SOURCE como comodín de destino
- Acepta MPI_ANY_TAG como comodín de tag

Recibir (síncrono)

- int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
  - buf - dirección inicial del buffer para enviar → output
  - count - máximo número de elementos en el buffer
  - dtyp - tipo de cada elemento en el buffer
  - source - rango mpi del destino (0..Rank_size-1)
  - tag - etiqueta para identificar el mensaje
  - comm - comunicador
  - status - estado → out

  inputs
- Se pueden recibir un máximo de count elementos, el número recibido se puede consultar usando MPI_Get_count
- Acepta MPI_ANY_SOURCE como comodín de origen
- Acepta MPI_ANY_TAG como comodín de tag

Send/receive ejemplo ping pong

https://github.com/wesleykendall/mpitutorial/blob/gh-pages/tutorials/mpi-send-and-receive/code/ping_pong.c

Para ser ejecutado con número par de procesos

```
int ping_pong_count = 0;
int partner_rank = (world_rank + 1) % 2;  → A quien envío
while (ping_pong_count < PING_PONG_LIMIT)
{
  if (world_rank == ping_pong_count % 2) → en una iteración envío y en la siguiente recibo
  { // Increment the ping pong count before you send it
    ping_pong_count++;
    MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD);
    printf("%d sent and incremented ping_pong_count " "%d to %d\n", world_rank, ping_pong_count, partner_rank);
  } else {
     MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("%d received ping_pong_count %d from %d\n", world_rank, ping_pong_count, partner_rank);
  }
}
```

Comunicación asíncrona

- Recibimos un handler al enviar/recibir el cual nos permite consultar el estado de la comunicación (finalizada no finalizada) y sincronizarnos
- int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest,
- int tag, MPI_Comm comm, **MPI_Request *request**)
- int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, **MPI_Request *request**)
  - **Request** es un parámetro de salida con el handler asociado a la petición
  - No asocia send con receive, solo afecta a cada petición

Sincronización

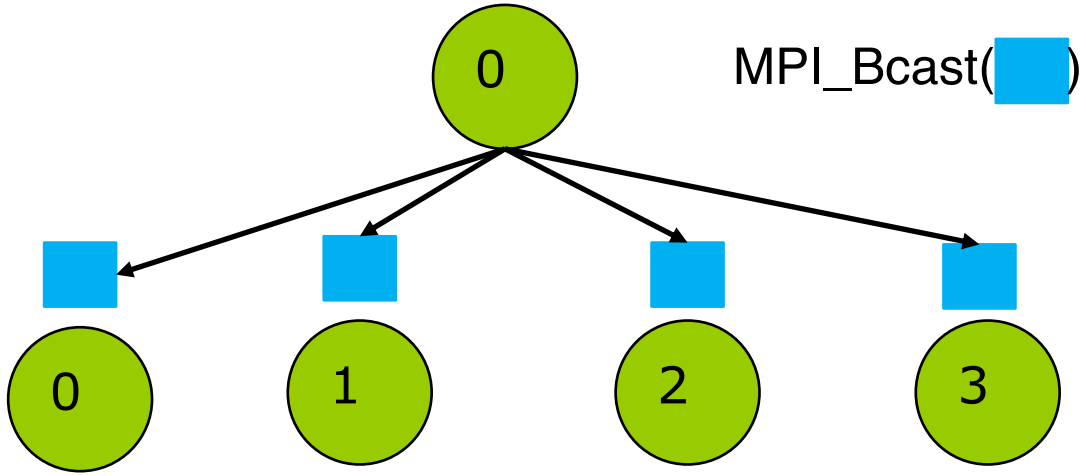
- Una vez tenemos un handler=request, podemos usar las siguientes funciones para esperar o testear el estado de la transferencia
  - int MPI_Wait(MPI_Request *request, MPI_Status *status);
  - int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index, MPI_Status *status);
  - int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
  - int MPI_Testany(int count, MPI_Request array_of_requests[], int *index, int *flag, MPI_Status *status);

Sincronizaciones globales

- int MPI_Barrier (MPI_Comm comm)
- Bloquea al proceso que lo llama hasta que todos los procesos del grupo entran en el barrier.

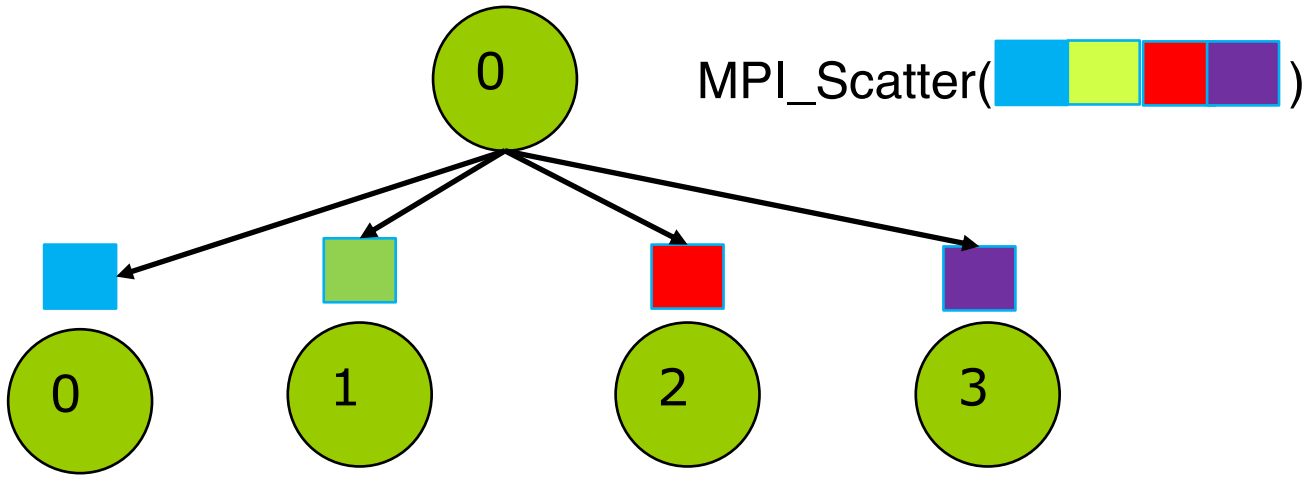
Colectivas:Broadcast

- El proceso con Rank=root envía **la misma información** a todos los procesos
- **Síncrono**: int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype, int **root**, MPI_Comm comm)
- **Asíncrono**: int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm, MPI_Request *request)



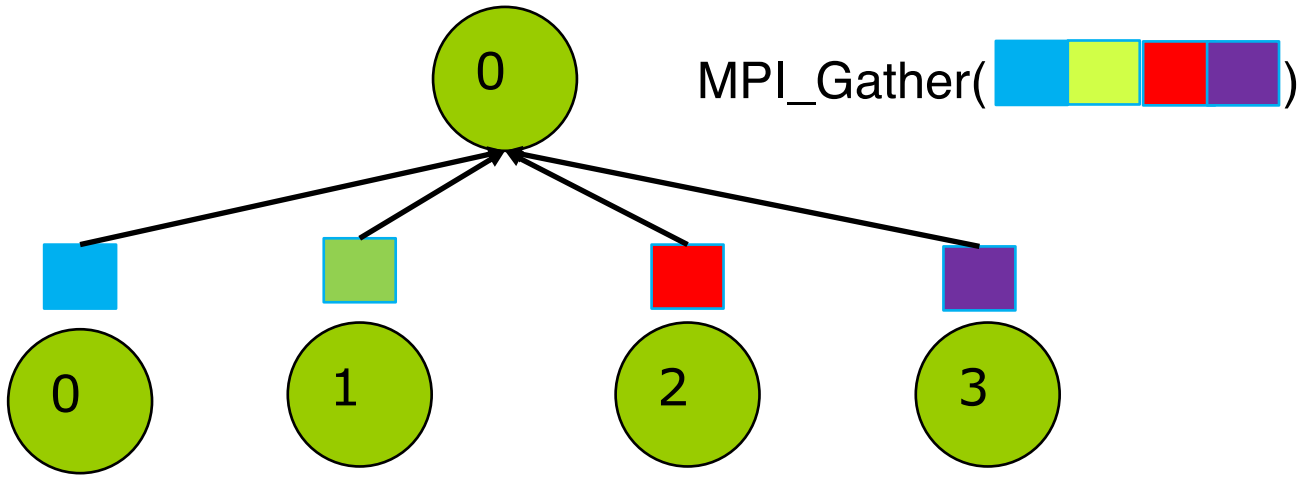
Colectivas: Scatter

- Un proceso distribuye un conjunto de datos entre un grupo de procesos. Cada proceso recibe un chunk
- **Síncrona**: int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- **Asíncrona**: int MPI_Iscatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)



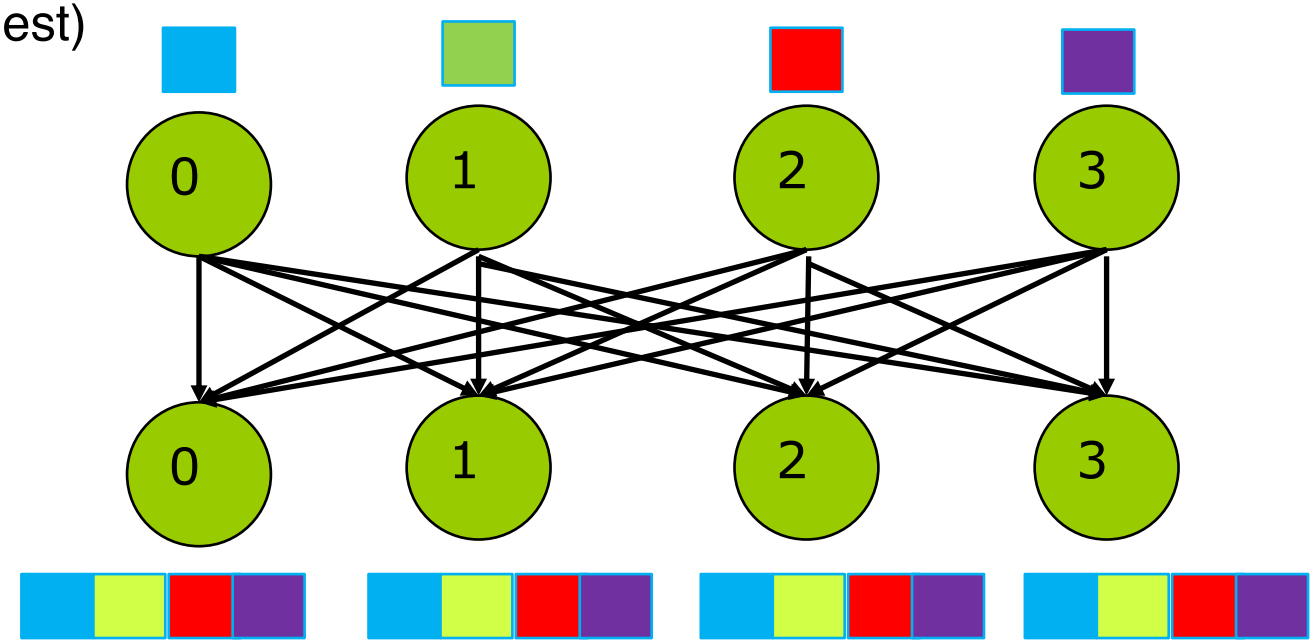
Colectivas: Gatter

- El gatter es la operación contraria al scatter, un proceso “recoge” un chunck de cada proceso del grupo
- **Síncrona**: int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- **Asíncrona**: int MPI_Igather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Request *request)



Colectiva: Allgather

- Los procesos intercambian información. Envían una parte y reciben del resto del grupo
- **Síncrono**: int MPI_Allgather (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
- **Asíncrono**: int MPI_Iallgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm, MPI_Request *request)



Colectivas con operaciones asociadas

- Versiones síncronas y asíncronas (añadir I , y request)

- int MPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  - Reduce todos los valores 1 solo
- int MPI_Allreduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
  - Reduce todos los valores a 1 solo y envía el resultado a todos
- La lista de operaciones está predefinida. Por ejemplo (consultar el manual para tener la lista completa):
  - MPI_MAX maximum
  - MPI_MIN minimum
  - MPI_SUM sum
  - MPI_PROD product
  - MPI_LAND logical and
  - etc

Ejemplo: MPI_Reduce