Seminario 3

Introducción a paso de mensajes con MPI

Sección 1. MPI

El esquema de funcionamiento de MPI implica un número fijo de procesos que se comunican mediante llamadas a funciones de envío y recepción de mensajes.

- SPMD: un programa, muchos datos.
- Permite también MPMD: muchos programas, muchos datos. Cada proceso puede ejecutar un programa diferente.
- En OpenMPI, la creación e inicialización de procesos sería como sigue:
  - **mpirun -oversubscribe -np 4 -machinefile maquinas prog1_mpi_exe**
  - Este comando comienza 4 copias del ejecutable.
  - El archivo maquinas define la asignación de procesos a ordenadores del sistema distribuido.
- Hay que hacer: **#include <mpi.h>**: define constantes, tipos de datos y prototipos de las funciones MPI.
- Las funciones devuelven código de error.
  - **MPI_SUCCESS**: Ejecución correcta.
- **MPI_Status** es un tipo de estructura con los metadatos de los mensajes:
  - **status.MPI_SOURCE**: proceso fuente.
  - **status.MPI_TAG**: etiqueta del mensaje.
- Constantes para representar tipos de datos básicos de C/C++ (para los mensajes de MPI): **MPI_CHAR, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED, MPI_UNSIGNED_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE**, etc.
- **Comunicador**: es tanto un grupo de procesos como un contexto de comunicación. Todas las funciones de comunicación necesitan como argumento un comunicador.

Sección 2. Compilación y ejecución de programas MPI

Tenemos varios scripts para trabajar con los programas:

- **mpicxx**: compila y/o enlaza programas C++ con MPI.
- **mpirun**: ejecuta programas MPI
- Ejemplos de compilación:
  - mpicxx -std=c++11 -c ejemplo.cpp
  - mpicxx -std=c++11 -o ejemplo_mpi_exe ejemplo.o
- También podemos compilar ditectamente:
  - mpicxx -std=c++11 -o ejemplo_mpi_exe ejemplo.cpp
- Para ejecutar un programa:
  - mpirun -oversubscribe -np 4 ./ejemplo_mpi_exe
  - El argumento **-np** sirve para indicar cuantos procesos ejecutarán el programa ejemplo, en este caso, cuatro.

    - Como no ponemos **-machinefile**, los cuatro procesos se lanzarán en el mismo ordenador que ejecuta **mpirun**.
    - La opción **-oversubscribe** puede ser necesaria si el número de procesadores disponibles en algún ordenador es inferior al número de procesos que se quieren lanzar en ese ordenador.

Sección 3. Funciones MPI básicas

Hay 6 funciones básicas en MPI:

- **MPI_Init**: inicializa el entorno de ejecución de MPI.
- **MPI_Finalize**: finaliza el entorno de ejecución de MPI.
- **MPI_Comm_size**: determina el número de procesos de un comunicador.
- **MPI_Comm_rank**: determina el identificador del proceso en un comunicador.
- **MPI_Send**: operación básica para envío de un mensaje.
- **MPI_Recv**: operación básica para recepción de un mensaje.

Para inicializar y finalizar un programa MPI se usan estas dos sentencias:

- **int MPI_Init(int *argc, char ****argv)
    - Se llama antes que cualquier otra función MPI.
    - Si se llama más de una vez durante la ejecución da un error.
    - Los argumentos son los argumentos de la línea de orden del programa.
- **int MPI_Finalize( )**
    - Llamado al fin de la ejecución.
    - Realiza tareas de limpieza para finalizar el entorno de ejecución.

Subsección 3.1. Introducción a los comunicadores

Un Comunicador MPI es un variable de tipo **MPI_Comm**. Está constituido por:

- Grupo de procesos.
- Contexto de comunicación: ámbito de paso de mensajes donde se comunican esos procesos. Un mensaje enviado en un contexto sólo puede ser recibido en dicho contexto.

MPI_COMM_WORLD hace referencia al comunicador universal, está predefinido e incluye todos los procesos lanzados:

- Un proceso puede pertenecer a diferentes comunicadores.
- Cada proceso tiene un identificador: desde 0 a P-1 (P es el número de procesos del comunicador).
- Mensajes destinados a diferentes contextos de comunicación no interfieren entre sí.

Para consultar el número de procesos tenemos la función **MPI_Comm_size**:

- **int MPI_Comm_size(MPI_Comm comm, int *size)**
- Escribe en el entero apuntado por size el número total de procesos que forman el comunicador comm.
- Si usamos el comunicador universal, podemos saber cuantos procesos en total se han lanzado en un aplicación.

Para consultar el identificador del proceso, tenemos la función **MPI_Comm_rank**:

- **int MPI_Comm_rank (MPI_Comm comm, int *rank)**

- Escribe en el entero apuntado por rank el número de proceso que llama. Este número es el número de orden dentro del comunicador comm.
- Se suele usar con el comunicador universal para identificar a cada proceso.

Subsección 3.2. Funciones básicas de envío y recepción de mensajes

El envío asíncrono seguro de un mensaje se puede conseguir con **MPI_Send**:

- **int MPI_Send(void *buf_emi, int num, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**
- Envía los datos (**num** elementos de tipo **datatype** almacenados a partir de **buf_emi**) al proceso **dest** dentro del comunicador **comm**.
- El entero **tag** se transfiere junto con el mensaje, y suele usarse para clasificar los mensajes en distintas categorías o tipos, en función de sus etiquetas. Es no negativo.
- Implementa envío asíncrono seguro, porque...
    - MPI ya ha leído los datos de **buf_emi**, y los ha copiado a otro lugar **al acabar MPI_Send**, por tanto podemos volver a escribir sobre **buf_emi** (envío seguro).
    - el receptor no necesariamente ha iniciado ya la recepción del mensaje(envío asíncrono).

La recepción segura síncrona de un mensaje se consigue usando **MPI_Recv**:

- **int MPI_Recv(void *buf_rec, int num, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)**
- Espera hasta recibir un mensaje del proceso **source** dentro del comunicador **comm** con la etiqueta **tag**, y escribe los datos en posiciones contiguas desde **buf_rec**.
- Puesto que se espera a que el emisor envíe, es una recepción síncrona. Puesto que al acabar ya se pueden leer en **buf_rec** los datos transmitidos, es una recepción segura.
- Valores especiales:
    - Si source es **MPI_ANY_SOURCE**, se puede recibir un mensaje de cualquier proceso en el comunicador.
    - Si tag es **MPI_ANY_TAG**, se puede recibir un mensaje con cualquier etiqueta.
- Los datos se copian desde **buf_emi** hacia **buf_rec**.
- Los argumentos **num** y **datatype** determinan la longitud en bytes del mensaje. El objeto **status** es una estructura con el emisor (**MPI_SOURCE**), la etiqueta (campo **MPI_TAG**).

Podemos obtener la cuenta de valores recibidos usando **status**:

- **int MPI_Get_count(MPI_Status *status, MPI_Datatype dtype, int *num)**
- Escribe en el entero apuntado por **num** el número de ítems recibidos en una llamada **MPI_Recv** previa. El receptor debe conocer y proporcionar el tipo de los datos **dtype**.

Sección 4. Paso de mensajes síncrono en MPI

Existe una función para envío síncrono (siempre es seguro):

- **int MPI_Ssend(void* buf_emi, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)**
- Inicia un envío, lee datos y espera el inicio de la recepción, con los mismos argumentos que MPI_Send.
- Es síncrono y seguro. Tras acabar **MPI_Ssend**:

    - ya se ha iniciado en el receptor una operación de recepción que encaja con este envío (síncrono).
    - los datos ya se han leído de **buf_emi** y se han copiado a otro lugar. Por tanto se puede volver a escribir en **buf_emi** (es seguro).
- Si la correspondiente operación de recepción usada es **MPI_Recv**, la semántica del paso de mensajes es puramente síncrona (hay una cita entre emisor y receptor).

Sección 5. Sondeo de mensajes

MPI incorpora dos operaciones que permiten a un proceso receptor averiguar si hay algún mensaje pendiente de recibir (en un comunicador), y en ese caso obtener los metadatos de dicho mensaje. Esta consulta:

- no supone la recepción del mensaje.
- se puede restringir a mensajes de un emisor.
- se puede restringir a mensajes con una etiqueta.
- cuando hay mensaje, permite obtener los metadatos: emisor, etiqueta y número de ítems (el tipo debe ser conocido).

Estas operaciones son:

- **MPI_Iprobe**: consultar si hay o no algún mensaje pendiente en este momento.
    - Es una consulta no bloqueante.
    - **int MPI_Probe (int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)**
    - Al terminar, el entero apuntado por flag será mayor que 0 solo si hay algún mensaje enviado al proceso que llama, y que encaje con los argumentos dentro del comunicador. Si no hay mensajes el entero es 0.
        - La consulta se refiere a los mensajes pendientes en el momento de la llamada.
        - Los parámetros (excepto **flag**) se interpretan igual que en **MSI_Probe**.
- **MPI_Probe**: esperar bloqueado hasta que haya al menos un mensaje.
    - **int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_STatus *status)**
    - Queda bloqueado el llamante hasta que haya al menos un mensaje enviado a dicho proceso en el comunicador **comm** y que encaje con los argumentos.
        - **source** puede ser un identificador de emisor o **MPI_ANY_SOURCE**.
        - **tag** puede ser una etiqueta o bien **MPI_ANY_TAG**.
        - **status** permite conocer los metadatos del mensaje, igual que se hace tras MPI_Recv.
        - si hay más de un mensaje disponible, los metadatos se refieren al primero que se envio.

Sección 6. Comunicación insegura

MPI ofrece la posibilidad de usar operaciones inseguras (asíncronas). Permiten el inicio de una operación de envío o recepción, y después el emisor o el receptor puede continuar su ejecución de forma concurrente con la transmisión:

- **MPI_Isend**: inicia envío pero retorna antes de leer el buffer.

    - **int MPI_Isend (void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)**
- **MPI_Irecv**: inicia recepción pero retorna antes de recibir.
    - **int MPI_Irecv (void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)**

Se puede comprobar posteriormente si la operación ha terminado o no:

- **MPI_Wait**: espera bloqueado hasta que acabe el envío o recepción.
    - Sirve para esperar bloqueado hasta que termine una operación.
        - **int MPI_Wait (MPI_Request *request, MPI_Status *status)**
- **MPI_Test**: comprueba si el envío o recepción ha finalizado o no. No supone espera bloqueante.
    - Comprueba la operación identificada por un ticket (**request**) y escribe en **flag** un número > 0 si ha acabado, o bien 0 si no ha acabado:
        - **int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)**
- **request** es un ticket que permitirá después identificar la operación cuyo estado se pretende consultar o se espera que finalice.
- Tanto en test como en wait una vez terminada la operación referenciada por el ticket:
    - podemos usar el objeto **status** para consultar los metadatos del mensaje.
    - la memoria usada por request es liberada por MPI.
- La recepción no incluye argumento **status** (se obtiene con las operaciones de consulta de estado de la operación).
- Cuando ya no se va a usar una variable **MPI_Request**, se puede liberar la memoria que usa con **MPI_Request_free** (NO HACE FALTA):
    - **int MPI_Request_free (MPI_Request *request)**