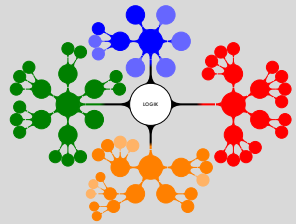
Seminario 3

Introducción a la programación con MPI

Asignatura *Sistemas Concurrentes y Distribuidos*

Fecha 19 noviembre 2019

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada

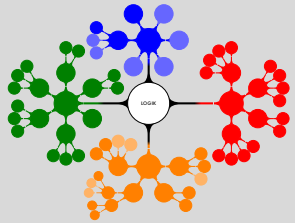


Objetivos

- El objetivo de esta práctica es familiarizar al alumno con el uso de la interfaz de paso de mensajes MPI y la implementación OpenMPI de esta interfaz
- Se indicarán los pasos necesarios para compilar y ejecutar programas usando OpenMPI
- Se presentarán las principales características de MPI y algunas funciones básicas de comunicación entre procesos.

Enlaces de interés:

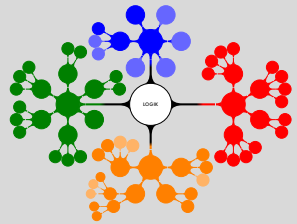
- `http://www.open-mpi.org/` Web Oficial de OpenMPI
- `http://lsi.ugr.es/jmantas/pdp/ayuda/instalacion.php?ins=openmpi` Instalación OpenMPI
- `http://lsi.ugr.es/jmantas/pdp/ayuda/mpi_ayuda.php` Ayuda para las funciones de MPI
- `http://lsi.ugr.es/jmantas/pdp/tutoriales/tutorial_mpi.php` Tutorial MPI



Configuración de `openmpi`

Para instalar `Openmpi` correctamente:

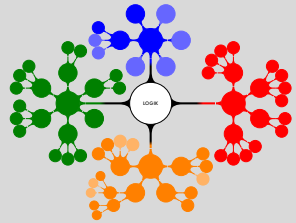
1. Bajar el comprimido de:
   `https://www.open-mpi.org/software/ompi/v4.0/`
   (openmpi-4.0.2.tar.gz)
2. Hacer (en Linux): `tar -xvf openmpi-*` y cambiarse `cd openmpi-4.0.2`
3. `./configure - -prefix=/home/$USER/.openmpi`
4. `make install`
5. Hay que incluir en nuestro entorno el directorio de instalacion (~ .openmpi) acabado en '/bin:'
   `export PATH="$PATH:/home/$USER/.openmpi/bin"`
6. Hay que incluir el directorio_de_instalacion_lib:
   `export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/$USER/.openmpi/lib/"`
7. Para hacerlos permanentes, teclear: `echo export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/home/$USER/.openmpi/lib/"» /home/$USER/.bashrc` y
   `echo export PATH="$PATH:/home/$USER/.openmpi/bin"»/home/$USER/.bashrc`



MPI

MPI es un estándar que define una API para programación paralela mediante paso de mensajes, que permite crear programas portables y eficientes

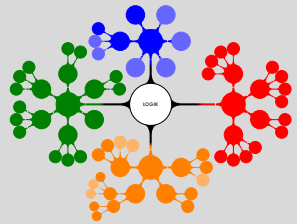
- Proporciona un conjunto de funciones que pueden ser utilizadas en programas escritos en C, C++, Fortran y Ada
- MPI-2 contiene más de 150 funciones para paso de mensajes y operaciones complementarias (con numerosos parámetros y variantes)
- Muchos programas paralelos se pueden construir usando un conjunto reducido de dichas funciones (hay 6 funciones básicas)



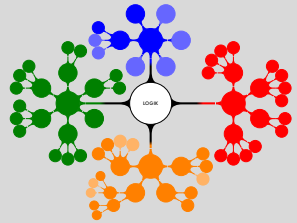
Modelo de programación MPI

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

- El modelo básico es SPMD (Single Program Multiple Data): todos los procesos ejecutan un mismo programa
- Permite el modelo MPMD (Multiple Program Multiple Data): cada proceso puede ejecutar un programa diferente
- La creación e inicialización de procesos no está definida en el estándar, depende de la implementación.
- En OpenMPI sería:`mpirun -np 4 -machinefile maquinas prog1`
  - Comienza 4 copias del ejecutable prog1.
  - El archivo maquinas define la asignación de procesos a ordenadores del sistema distribuido



Compilación de programas en `Openmpi`



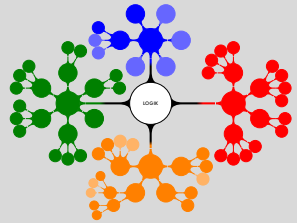
OpenMPI es una implementación portable y de código abierto del estándar MPI-2, llevada a cabo por una serie de instituciones de ámbito tanto académico y científico como industrial.
OpenMPI ofrece varios scripts necesarios para trabajar con programas aumentados con llamadas a funciones de MPI. Los más importantes son estos dos:

- `mpicxx`: para compilar y/o enlazar programas C++ que hagan uso de MPI
- `mpirun`: para ejecutar programas MPI. El programa `mpicxx` puede utilizarse con las mismas opciones que el compilador de C/C++ usual, p.e.:
  - `$ mpicxx -std=c++11 -c ejemplo.cpp`
  - `$ mpicxx -std=c++11 -o ejemplo ejemplo.o`

Compilación de programas en `Openmpi` -II

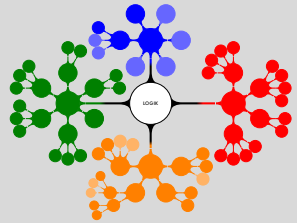
La forma más usual de ejecutar un programa MPI es :

- `$ mpirun -np 4 ./ejemplo`
- El argumento `-np` sirve para indicar cuántos procesos ejecutarán el programa ejemplo. En este caso, se lanzarán cuatro procesos ejemplo
- Como no se indica la opción `-machinefile`, OpenMPI lanzará dichos 4 procesos en el mismo ordenador donde se ejecuta `mpirun`
- Con la opción `machinefile`, podemos realizar asociaciones de procesos a distintos ordenadores
- Si al ejecutar `mpirun` aparece el error: There are not enough slots available in the system . . .:
  - Crearse un archivo *hostfile* en el directorio de trabajo que contenga 1 sola línea con `localhost slots=40` (o un número mayor de slots necesarios para `-np`)
  - Cambiar la ejecución del programa a: `$ mpirun -hostfile hostfile -np 4 ./ejemplo`



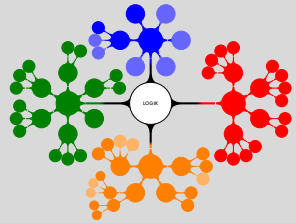
Implementación con OpenMPI

- Hay que escribir: `#include < mpi.h >`: define constantes, tipos de datos y los prototipos de las funciones MPI
- Las funciones devuelven un código de error. MPI_SUCCESS (corresponde con el valor '0'): Ejecución correcta
- MPI_Status es un tipo estructura con los metadatos de los mensajes:
  - status.MPI_SOURCE: proceso origen del mensaje
  - status.MPI_TAG: etiqueta del mensaje
  - Constantes para representar tipos de datos básicos de C/C++ (para los mensajes en 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

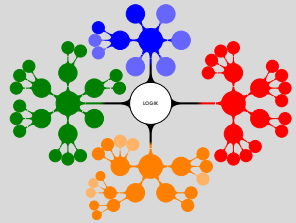


Funciones MPI básicas

- 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



Inicializar y finalizar el programa



Se usan dos sentencias:

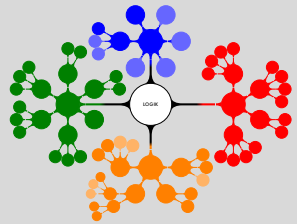
- `int MPI_Init( int *argc, char ***argv )`
  - Llamado antes de cualquier otra función MPI
  - Si se llama más de una vez durante la ejecución da un error
  - Los argumentos argc, argv son los argumentos de la línea de orden del programa
- `int MPI_Finalize( )`
  - Llamado al final de la computación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

Consulta del número de procesos

La función MPI_Comm_size tiene esta declaración:

- 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 una aplicación, por ejemplo:

```
1 int num_procesos ; // contendra el total de procesos de ↩
      la aplic.
2 MPI_Comm_size( MPI_COMM_WORLD, &num_procesos );
3 cout << "El numero total de procesos es: " << ↩
      num_procesos << endl ;
```

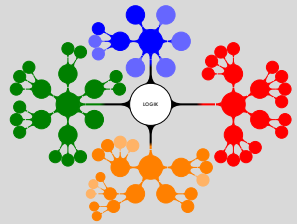


Consulta del identificador de procesos

La función MPI_Comm_rank tiene esta declaración:

- Escribe en el entero apuntado por rank el número de proceso que llama. Este número es el número de orden dentro del comunicador comm (empezando en 0). Ese número se suele llamar rank o identificador del proceso en el comunicador
- Se suele usar al inicio de una aplicación MPI, con el comunicador universal, como en el ejemplo siguiente

```
1 int id_propio ; // ácontendr el únmero de proceso que ↩
      llama
2 MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
3 cout << "mi únmero de proceso es:" << id_propio << endl ↩
      ;
```



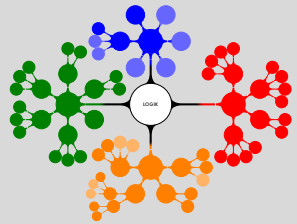
Ejemplo de un programa simple

En el archivo `holamundo.cpp` vemos un ejemplo sencillo:

si se compila y ejecuta con 4 procesos, obtendremos la una salida similar a la siguiente:

```
1 Hola desde proc. 0 de 4
2 Hola desde proc. 2 de 4
3 Hola desde proc. 1 de 4
4 Hola desde proc. 3 de 4
```

```
1  include <mpi.h>include <iostream>
2  using namespace std;
3  int main( int argc, char *argv[] ){
4  int id_propio, num_procesos ;
5  MPI_Init( &argc, &argv );
6  MPI_Comm_size( MPI_COMM_WORLD, &num_procesos );
7  MPI_Comm_rank( MPI_COMM_WORLD, &id_propio );
8  cout <<"Hola desde proceso " <<id_propio <<" de " <<↩
       num_procesos <<endl;
9  MPI_Finalize();
10 return 0;}
```

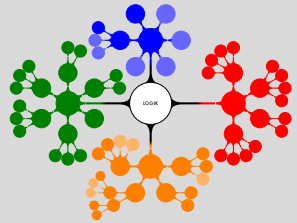


Envío asíncrono seguro de un mensaje (MPI_Send)

Un proceso puede enviar un mensaje usando MPI_Send:

```
1 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 (etiqueta) 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 envio asíncrono seguro: tras acabar MPI_Send
  - MPI ya ha leído los datos de buf_emi, y los ha copiado a otro lugar, por tanto podemos volver a escribir sobre buf_emi (el envío es seguro)
  - el receptor no necesariamente ha inciado ya la recepción del mensaje (el envío es asíncrono)

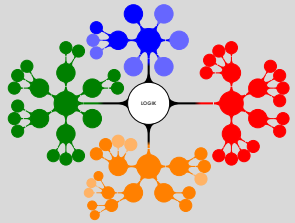


Recepción segura síncrona de un mensaje (MPI_Recv)

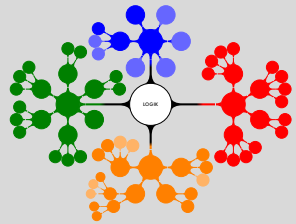
Un proceso puede recibir un mensaje usando MPI_Recv, que se declara como sigue:

```
1 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
- Se pueden dar valores especiales o comodín:
  - 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



Obtener la cuenta de valores recibidos con MPI_Get_Count



Los argumentos num y datatype determinan la longitud en bytes del mensaje. El objeto status es una estructura con el emisor (campo MPI_SOURCE), la etiqueta (campo MPI_TAG)

- Para obtener la cuenta de valores recibidos, usamos `status`

```
1 int MPI_Get_count( MPI_Status *status, MPI_Datatype↩
      dtype, int *num );
```

- Escribe en el entero apuntado por num el número de items recibidos en una llamada MPI_Recv previa. El receptor debe conocer y proporcionar el tipo de los datos (dtype)