1. Introducción. Convenciones

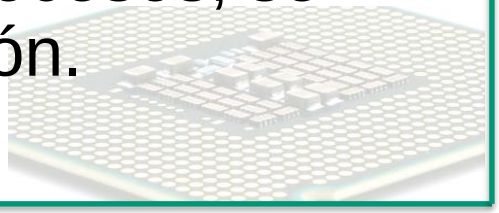
- Convenciones en cualquier programa MPI:
  - Incluir el fichero de **cabecera**: con los prototipos, tipos de datos, variables y constantes.
  - Las llamadas a MPI en forma de funciones en ANSI C empiezan con "**MPI_**" seguido de una mayúscula y minúsculas. Igual los tipos de datos.
  - Las llamadas a MPI en FORTRAN empiezan con "MPI_" seguido de mayúsculas.
  - Todas las **constantes** están en mayúsculas.
  - Las funciones en C devuelven un **entero** de diagnóstico (en FORTRAN a través de un argumento), el éxito está definido como una constante: MPI_SUCCESS.

1. Introducción. Estructura

- MPI implementa el modelo **SPMD** (Single Program Multiple Data) equivalente al **fork**.
  - Todos ejecutan un mismo programa y dependiendo de su identificación hacen una cosa u otra.

```
if (pid == 1) ENVIAR_a_pid2
else if (pid == 2) RECIBIR_de_pid1
```

  - Cada proceso tiene su propio espacio de memoria (no es compartido).
- MPI 1 tiene una gestión estática de procesos. Se da el número al arrancar (ver posteriormente).
- MPI 2 tiene gestión dinámica de procesos, se pueden crear en tiempo de ejecución.
  - También puede hacer MPMD / MIMD.



1. Introducción. Estructura

MPI include file

Declarations, prototypes, etc.

Program Begins

Serial code

Initialize MPI environment — *Parallel code begins*

Do work and make message passing calls

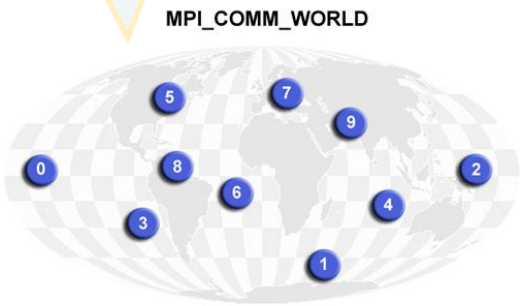
Terminate MPI Environment — *Parallel code ends*

Serial code

Program Ends

Probarlo con ejemplo

Número de procesos que se ejecutan el código paralelo



The MPI Standard does not say what a program can do before an **MPI_Init** or after an **MPI_Finalize**. In the Open MPI implementation, it should do as little as possible.

1. Introducción. Básico en C

- Las cuatro funciones básicas de MPI:
  - `int MPI_Init(int *argc, char ***argv);`
    - Inicializa la aplicación paralela (se pueden tomar datos de la línea de comando en ANSI C).
  - `int MPI_Comm_size (MPI_Comm comm, int *size);`
    - Obtiene el número de procesos de un comunicador.
  - `int MPI_Comm_rank (MPI_Comm comm, int *rank);`
    - Obtiene la identificación del proceso que hace la llamada (desde cero hasta el número de procesos menos 1).
  - `int MPI_Finalize(void);`
    - Finaliza la aplicación paralela (puede seguir la secuencial).

- MPI_Comm es un tipo de dato que indica el comunicador (grupo de procesos). MPI_COMM_WORLD es una constante que indica el comunicador global. Se verán posteriormente con más detalle.

1. Introducción. Ejemplo

- Un ejemplo sencillo (en C):

```
#include <mpi.h>
#include <stdio.h>
main (int argc, char **argv)
{
  int nproc; /* Número de procesos */
  int yo; /* Mi dirección: 0<=yo<=(nproc-1) */
  puts("empieza la parte paralela");
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &yo);
  printf("Numero de procesos %d,
                          yo soy %d\n", nproc, yo);
  MPI_Finalize();
  puts("termina la parte paralela");
  puts("sigue la parte secuencial");
}
```

Lo que se puede probar

1. Introducción. Ejemplo

- Un ejemplo sencillo (en FORTRAN):

```
program main
include 'mpif.h'
integer ierr
C Inicio MPI
call MPI_INIT( ierr )
print *, 'Hola Mundo!'
call MPI_FINALIZE( ierr )
C Termino MPI
end
```

1. Introducción. Compilación y Ejecución

- Compilación en línea:
  - Genérico:
    - mpicc
    - mpif77
  - Sistemas Linux:
    - g++ -o programa programa.C -lmpi++ -lmpi
    - gcc -o programa programa.c -lmpi
  - Sistemas IRIX para 64 bits:
    - cc -64 programa.c -lmpi
    - f77 -64 -LANG:recursive=on programa.f -lmpi
    - f90 -64 -LANG:recursive=on programa.f -lmpi
    - CC -64 programa.C -lmpi++ -lmpi
- Proyectos grandes: mejor usar la herramienta make. Y colaborativos GitHub.

1. Introducción. Compilación y Ejecución

- Ya tenemos el ejecutable, ahora viene la ejecución del mismo:
  - Se necesita un programa que lance N copias del mismo ejecutable (que pueden hacer instrucciones distintas a través de selección).
  - Cada copia (proceso) avanza independientemente y tiene su propio espacio de memoria.
  - La sincronización y la comunicación entre los procesos se realiza explícitamente a través de MPI.
- Esto se consigue con un lanzador (mpirun):
  - `mpirun -np 3 programa argumento1 argumento2`
    - Se le indica el número de copias (procesos / procesadores), el programa a copiar, y sus posibles argumentos (como un programa en C).

1. Introducción. Compilación y Ejecución

- Existe un equivalente del mpirun que es mpiexec.
- Además se puede definir de forma explícita donde vamos a ejecutar nuestro programa de dos formas:
  - En línea usando la opción –H:
    - `mpirun -H aa,aa,bb a.out`
    - `mpirun -H aa,bb -npernode 2 a.out`
  - Con el fichero de hosts:
    - `mpirun -hostfile myhostfile -np 6 a.out`

    donde el fichero puede ser:

    ```
    aa slots=4
    bb slots=4
    cc slots=4
    ```

Práctica 0

- En el laboratorio se usa open MPI.
  - Se puede usar de dos maneras:
    - En un solo nodo con dos cores.
    - En todo el laboratorio: Varios nodos encendidos (2 cores) unidos por red.
  - El ORTE usado es con ssh.
    - Hay que generar la clave pública: `ssh-keygen -t rsa`
    - Esto c… directorio "`.ssh`" en el que habrá dos ficheros nuevos "`id_r…ub`". Es aconsejable dejar vacía la password.
    - Copia… `pub`) al fichero de autorizaciones: "`aut…`"
    - A pa… …uiera de las máquinas.
      - La primera v… máquina quedará registrada en "`known_…`"
    - Se puede usar `for IP in `mac… …sh $IP echo; done` para poner los "yes" una vez y listo.
    - En cada sesión que realicemos además:
      - Se arranca el agente ssh : `eval `ssh-agent``
      - Se indica al agente la clave : `ssh-add $HOME/.ssh/id_rsa`
  - Existe un "comando" propio que nos da la lista (IP) de máquinas: "`machine_list`".
  - Se puede usar para generar el fichero de máquinas.
  - Problema: si trabajáis todos a la vez y lanzáis la carga no se va a ganar mucho. Sería mejor que lo hicierais de forma escalonada.

Ver ejercicios