Toma de contacto con MPI

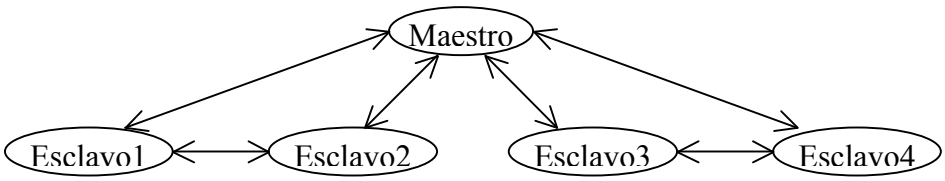
Septiembre 2010

ÍNDICE

1 Visión global de MPI.

MPI significa "**M**essage **P**assing **I**nterface" y procede del esfuerzo de la comunidad de la computación paralela (industria y grupos de desarrollo de software libre) que creó el MPI Forum con el objetivo de desarrollar un estándar para programación paralela basada en paso de mensajes. El primer estándar se publicó en 1994 y, posteriormente, se generó una segunda versión "MPI-2" publicada en 1997. A partir de estas definiciones existen implementaciones tanto comerciales como de libre distribución (nosotros utilizaremos **MPICH**). MPI es, junto con **PVM** "**P**arallel **V**irtual **M**achine", uno de los paquetes de software más populares en el entorno de aplicaciones basadas en paso de mensajes.

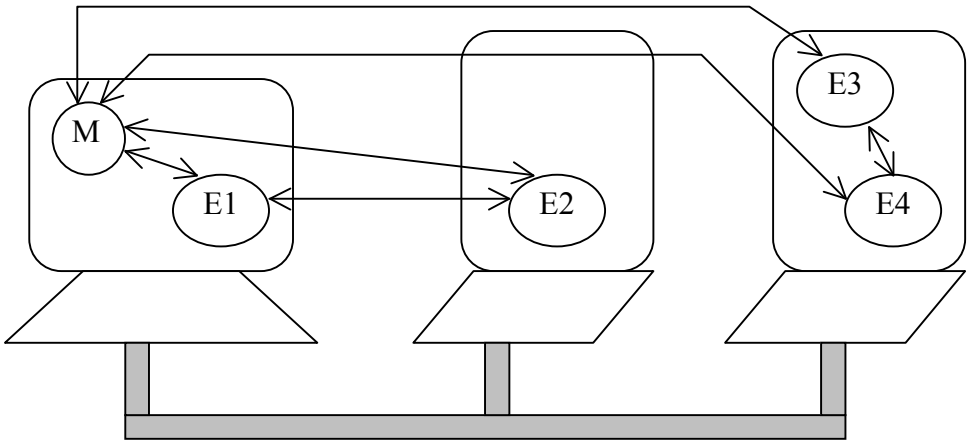
Una aplicación paralela típica sería aquella en la que un proceso maestro crea varios procesos esclavos para realizar, cada uno de ellos, una parte del trabajo global. Gráficamente, este modelo es el siguiente:



donde el Maestro se comunica con sus esclavos, el Esclavo1 se comunica con el Esclavo2 y el Esclavo3 con el Esclavo4.

Con un soporte de sistema operativo multitarea (como UNIX), esta aplicación podría ejecutarse en una máquina monoprocesador multiplexando la CPU entre los cinco procesos que se crearían y comunicándose bien a través de memoria común o por un mecanismo de paso de mensajes.

Si disponemos de más de una máquina física (por ejemplo PC's o iMac's o Supercomputadores) que tienen interconectividad entre sí a través de una red de datos, MPI nos permitirá contemplar dichas máquinas físicas como un conjunto de nodos de computación (cluster) en el que ejecutar nuestra aplicación paralela de tal forma que los procesos se distribuyan entre las distintas CPU's disponibles y así poder ejecutarse con mayor paralelismo real. Supuesto que tenemos un PC y dos iMacs, la aplicación ejemplo podría ejecutarse tal y como se muestra a continuación:



donde las comunicaciones entre los procesos que se ejecutan en máquinas distintas, fluirán a través de la red de datos.

Es importante entender que MPI es una biblioteca, no un nuevo lenguaje de programación. Esta biblioteca define primitivas que permiten: crear procesos, comunicarse entre ellos mediante el envío/recepción de mensajes, etc. Esta biblioteca se genera al instalar el entorno de MPI y está escrita en "C", si bien también existe interfaz para C++ y FORTRAN.

Algunas de las características de MPI son:

- Grupos dinámicos de procesos.
- Soporte para redes de ordenadores heterogéneos.
- Comunicación (síncrona y asíncrona) mediante envío y recepción de mensajes con posibilidad de multicast y broadcast.
- Integración con sistemas basados en multiprocesadores.

2 Primitivas básicas de MPI para crear procesos y comunicarse entre ellos.

Se recomienda consultar la referencia "*www-unix.mcs.anl.gov/mpi/www/*" para una descripción completa de las funciones que aquí explicaremos así como del resto de funciones de MPI por si fuesen de utilidad en algún caso.

Todas las funciones devuelven un valor "int" que sería el código de error. De todas formas, si no se hace nada especial, ante un error, el entorno provoca la terminación del proceso.

2.1 Control de procesos.

```
int MPI_Comm_size (MPI_Comm grupo, int *cuantos)
```

Devuelve en "cuantos" el número de procesos del "grupo" indicado. En general, siempre utilizaremos el grupo MPI_COMM_WORLD, en cuyo caso, lo que nos dirá esta función es con cuántos procesos se ha arrancado nuestra aplicación.

```
int MPI_Comm_rank (MPI_Comm grupo, int *yo)
```

Esta función devuelve en "yo" el identificador del proceso que la invoca dentro del "grupo" indicado. Los identificadores van del cero en adelante (hasta el número de elementos de dicho grupo). Se trata pues de una función similar a *getpid* de UNIX.

```
int MPI_Finalize ( )
```

Sirve para informar al entorno MPI de que este proceso termina. La terminación definitiva de un proceso debe hacerse de la forma siguiente:

```
MPI_Finalize ( );      // Para informar al soporte de MPI
exit (0);              // Para informar al Sistema Operativo
```

```
int MPI_Init  (int  *argc,  char ***argv)
```

Inicializa la ejecución del entorno MPI. Su efecto es arrancar los procesos indicados en la línea de comando. Se debe pasar un puntero al número de argumentos de línea de comando (parámetro *argc* del *main* de C) y otro puntero al vector de argumentos (parámetro *argv* del *main* de C).

Como ejemplo de uso de las cuatro funciones anteriores, está el código siguiente que se corresponde con el programa *helloWorld.c* listado completo en los ANEXOS.

```
int main(int argc, char *argv[]) {
  int yo, numProcesos;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &yo);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcesos);
  if (yo == 0) printf ("Se han creado %d procesos\n", numProcesos);
  printf ("Soy el proceso %d\n", yo);
  MPI_Finalize();
  return 0;
}
```

Si ejecutásemos el programa anterior con un comando como el siguiente:

```
%mpirun –all-local –np 5 helloWorld
```

donde **–np 5** indica que se arranquen cinco procesos que ejecuten todos ellos el programa *helloWorld* en la máquina local, la salida podría ser la siguiente:

```
Se han creado 5 procesos
Soy el proceso 0
Soy el proceso 1
Soy el proceso 2
Soy el proceso 4
Soy el proceso 3
```

2.2 Envío y recepción simple de mensajes.

Existe una gran variedad de primitivas de envío y recepción de mensajes. Para empezar, trataremos sólo con las dos primitivas más básicas de envío y recepción (*MPI_Send* y *MPI_Recv* respectivamente).

Tanto el envío como la recepción están orientados a la transferencia de arrays de datos. Si se desea enviar tan sólo un valor, por ejemplo un entero, puede entenderse como el envío de un array de un único elemento.

```
int MPI_Send (void *buffer, int cuantos, MPI_Datatype tipoDato,
              int destino, int etiqueta, MPI_Comm grupo)
```

El envío *puede* bloquear al proceso que hace el envío hasta que se haya recibido el mensaje. El bloqueo dependerá de la posibilidad de almacenar temporalmente el buffer en el caso de que el receptor no esté esperando el mensaje, y es dependiente de la implementación.

El significado de los parámetros es el siguiente:

- *buffer* Dirección del vector a enviar
- *cuantos* Número de elementos a enviar (igual o mayor que cero)
- *tipoDato* Tipo de los datos del vector (MPI_INT, etc. como se ve más adelante)
- *destino* Identificador del proceso destinatario
- *etiqueta* Etiqueta del mensaje. Debe estar en el rango 0..MPI_TAG_UB
- *grupo* Grupo de comunicación. En principio MPI_COMM_WORLD

Los valores posibles de MPI_Datatype son los siguientes:

MPI_Datatype	Tipo de dato en 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	

El tipo MPI_BYTE no se corresponde con un tipo en C y es un dato de 8 bits sin interpretación alguna.

```
int MPI_Recv (void *buffer, int cuantos, MPI_Datatype tipoDato,
              int origen, int etiqueta, MPI_Comm grupo,
              MPI_Status *estado)
```

El proceso se quedará bloqueado hasta recibir el mensaje indicado.

El significado de los parámetros es el siguiente:

- *buffer* Dirección del vector donde se dejará el mensaje
- *cuantos* Número máximo de elementos en el *buffer* de recepción. El número de elementos realmente recibidos puede conocerse con MPI_Get_count
- *tipoDato* Tipo de los datos del vector (los mismos que en MPI_Send)
- *origen* Identificador del proceso remitente. También puede indicarse MPI_ANY_SOURCE, en cuyo caso se expresa el deseo de recibir de cualquiera
- *etiqueta* Etiqueta del mensaje. También puede indicarse MPI_ANY_TAG, en cuyo caso se expresa el deseo de recibir un mensaje de cualquier etiqueta
- *grupo* Grupo de comunicación. En principio MPI_COMM_WORLD
- *estado* Resultado de la recepción. Se trata de una estructura que tiene, entre otros, los campos siguientes:
    - MPI_SOURCE que indica quién envió el mensaje. Útil en combinación con MPI_ANY_SOURCE
    - MPI_TAG que indica la etiqueta del mensaje recibido. Útil en combinación con MPI_ANY_TAG

```
int MPI_Get_count (MPI_Status *estado, MPI_Datatype tipoDato,
                   int *cuantos)
```

Devuelve en "cuantos" el número de elementos del tipo "tipoDato" recibidos por la operación de recepción que devolvió el "estado" indicado como primer parámetro.

```
int MPI_Bcast (void *buffer, int cuantos, MPI_Datatype tipoDato,
               int emisor, MPI_Comm grupo)
```

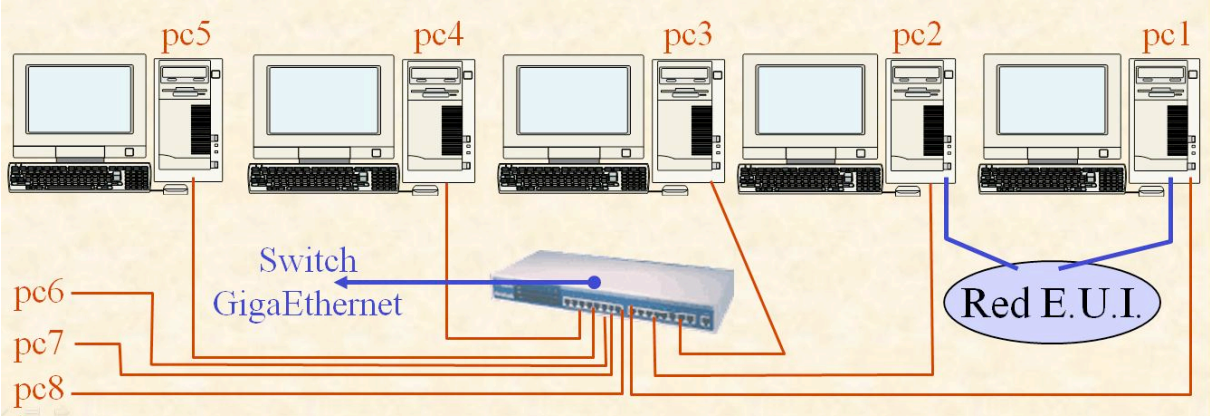
Esta primitiva envía un mensaje desde el proceso "emisor" al resto de procesos que pertenezcan al "grupo" indicado, en principio MPI_COMM_WORLD.

Hay que resaltar que esta función tiene que ser invocada tanto por el "emisor" (actuando la primitiva como un envío) como por los receptores (en cuyo caso la función actúa como una primitiva de recepción).

Los parámetros "buffer", "cuantos" y "tipoDatos" identifican el vector que se va a enviar en el caso del proceso "emisor", y el vector donde se recibirá la información en el caso de los receptores.

3 El cluster de prácticas.

El aspecto del equipo de prácticas (cluster formado por ocho PC's Quad) es el siguiente:



Puede observarse que hay dos máquinas, denominadas "iaclusters1" (pc1) e "iaclusters2" (pc2) que tienen dos interfaces de red, uno para conectarse a la red de la Escuela y otro para la Intranet formada por los cinco PC's de cuatro núcleos mediante un switch GigaEthernet.

Para acceder al cluster desde cualquier otro equipo de la Escuela, vía telnet, tendríamos que entrar en "iaclusters1.eui.upm.es" o "iaclusters2.eui.upm.es" y ya en uno de ellos, podríamos acceder al resto a través de la red interna.

Los nombres lógicos de "iaclusters1" e "iaclusters2" para el interfaz de la red interna son "pc1" y "pc2" respectivamente. Los otros seis PC's (denominados "pc3", "pc4" … "pc8") tan sólo tienen un interfaz de red activado y conectado al switch y no son accesibles directamente desde la red de la Escuela.

Los PC's 1 a 5 incorporan procesadores Intel Core 2 Quad Q6600 a 2,4 GHz con 8MB de caché L2 cada procesador, 4GB de memoria y 500GB de disco. Los PC's 6, 7 y 8 incorporan procesadores Intel Core i7 920 a 2,66GHz con 8MB de caché L3 compartida y 256KB de caché L2 por núcleo, 6GB de memoria y 500GB de disco.

Todos los grupos de prácticas tienen abierta cuenta en todos los equipos con login de nombre "arqava01", "arqava02" y sucesivos, con una clave de acceso única que se suministrará al iniciar la práctica. El servidor "iaclusters1" se ha configurado como servidor de páginas amarillas y mantiene los login y password de todas las cuentas de forma centralizada.

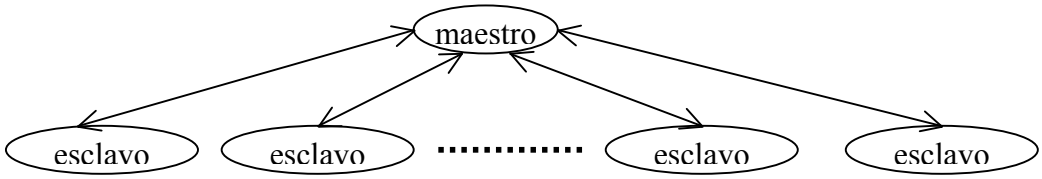
Está previsto que las prácticas se hagan en turnos de dos horas con cabida para ocho grupos de alumnos que utilizarán como equipo de entrada "pc1", "pc2", … y "pc8". Para no interferirse unos con otros, cada grupo dispondrá de cuatro núcleos en su propio equipo. En momentos puntuales, un grupo podrá incorporar los otros PC's para hacer pruebas con hasta treinta y dos núcleos trabajando para la ejecución de su programa.

Para entrar en un equipo del cluster desde otro equipo con UNIX, debe utilizarse el comando: "**ssh [-l login] maquina**", por ejemplo "ssh pc5" tecleado desde pc1 para entrar en pc5 con el mismo login que en pc1. Si el acceso es desde un PC con Windows, por ejemplo desde el Centro de Cálculo, utilizar la utilidad "**putty**" con el protocolo SSH (recordar que sólo habría acceso directo a iaclusters1 e iaclusters2).

Si queremos transferir información entre dos equipos del cluster, puede utilizarse la versión segura de "**ftp**" mediante el comando: "**sftp [login@]maquina**". Más adelante (ver sección 5), también presentaremos otro comando "**mpircopy**" basado en "**rcp**" que también permite hacer transferencia de un fichero al resto de máquinas del cluster. Si la transferencia es entre un PC con Windows desde el Centro de Cálculo y uno de los servidores del cluster, utilizar la utilidad "**WinSCP**".

4 El primer programa "hola" ejecutándose sobre una única máquina física.

Vamos a probar la ejecución de un programa de ejemplo que consiste en un proceso "maestro" que se comunica con "n" procesos "esclavos". Gráficamente:



El proceso "maestro" indicará dónde se ejecuta y se pondrá a esperar un mensaje de cada "esclavo" que le informa desde qué máquina se ejecuta. Después se pondrá a recibir de cada uno de ellos otro mensaje con el tiempo que han tardado en ejecutarse. De todo ello informará por la salida estándar.

Cada proceso "esclavo" lo que hará básicamente es una espera mediante "sleep", en concreto "sleep (yo + 5)", de forma que el primer proceso espere sólo 6 segundos, el siguiente 7 segundos y así sucesivamente.

Para poder probar todo esto, dispondremos del fichero "hola.c" (que contiene tanto el código del proceso "maestro" como el del "esclavo") y el correspondiente "Makefile" para facilitar la labor de compilación para todos los programas de esta práctica.

Ahora vamos a probar realmente todo esto en el ordenador, para ello:

- Entrar a un puesto con el *login* de la cuenta del grupo (arqava01, arqava02, etc. que se suministra en este momento) junto con la palabra clave asociada.
- Cambiar la palabra clave (comando "**yppasswd**").
- Crear el directorio $HOME/p1 (mkdir p1) para realizar esta práctica.
- Situarse en $HOME/p1 (*cd p1*).
- Copiarse los ficheros "hola.c", "Makefile" y demás, que se suministran para la realización de esta práctica. Todos ellos residen en el directorio:

  ../../arqava00/p1

- Teclear lo siguiente: "cp ../../arqava00/p1/* ." ¡No olvidar los puntos!
- Comprobar que se han copiado los ficheros siguientes:

  costemc.c costeop.c helloWorld.c hola.c holaf.txt
  Makefile pingpong.c pingpongf.txt psendrec.c

- Echar un vistazo al fichero "Makefile".

- Echar un vistazo al programa hola.c.
- Compilar el programa hola.c: make hola.
- Comprobar que se ha generado el ejecutable correspondiente "hola".
- Antes de ejecutar el programa hola, vamos a ver la ayuda del comando de ejecución de programas MPI. Para ello, teclear lo siguiente:

  mpirun –help

  De todas las opciones que tiene disponibles, tan sólo comentaremos "np", "p4pg", "stdout" y "stdin". La opción "np" ya se comentó en la sección 2.1 y permite indicar el número de procesos que queremos crear. En vez de utilizar esta opción, de momento utilizaremos "p4pg" tal y como veremos a continuación. Las opciones "stdout" y "stdin" permiten, respectivamente, desviar la salida y entrada estándar a un fichero, pero, en su lugar, utilizaremos las redirecciones ">" y "<" propias de UNIX.

- Cuando se indica la opción "p4pg" con el comando "mpirun", se está diciendo, a través de un fichero, cuántos procesos arrancar, en qué máquinas y qué programa ejecutarán. Para entender el formato de estos ficheros, echar un vistazo al fichero "holaf.txt" cuyo contenido debería ser parecido a:

  pc1 8 /usuarios/alumnosArqava/**arqava01**/p1/hola

  Este fichero, pensando que se trata del grupo "arqava01" con nodo de trabajo "pc1", indica que se arranquen 8 procesos en el nodo "pc1" y que, todos ellos, ejecuten el programa "hola". Adaptar el fichero a nuestro **PC** y **login**.
- Ahora vamos a lanzar la ejecución del programa "hola" con el comando **mpirun** tecleando lo siguiente:

  mpirun –p4pg holaf.txt hola

  Con esta forma de ejecución, **realmente se ejecutarán 9 procesos**: uno arrancado desde la consola, que ejecutará el programa hola como proceso 0 y otros 8 procesos que también ejecutarán "hola" en el "pc1" con identificadores de proceso del 1 al 8.

  Observaremos una salida como la que aparece en la página siguiente.

```
Maestro ejecutandose en pc1
Del proceso 1: Hola, desde pc1
Del proceso 4: Hola, desde pc1
Del proceso 6: Hola, desde pc1
Del proceso 2: Hola, desde pc1
Del proceso 5: Hola, desde pc1
Del proceso 3: Hola, desde pc1
Del proceso 8: Hola, desde pc1
Del proceso 7: Hola, desde pc1
Tiempo del Proceso[1] = 6:0 (seg:mseg)
Tiempo del Proceso[2] = 7:2 (seg:mseg)
Tiempo del Proceso[3] = 8:1 (seg:mseg)
Tiempo del Proceso[4] = 9:1 (seg:mseg)
Tiempo del Proceso[5] = 10:1 (seg:mseg)
Tiempo del Proceso[6] = 11:1 (seg:mseg)
Tiempo del Proceso[7] = 12:1 (seg:mseg)
Tiempo del Proceso[8] = 13:0 (seg:mseg)
P4 procgroup file is holaf.txt.
```

Intentar explicar por qué los primeros mensajes de "Hola … " desde los esclavos salen en desorden y los últimos de "Tiempo … " lo hacen en orden.

- Para realizar la misma ejecución desviando la salida a un fichero, teclear:

```
mpirun –p4pg holaf.txt hola > salida.txt
```

Comprobar que el fichero "salida.txt" tiene lo mismo que vimos por pantalla.

- **Avisar al profesor** para ver cómo depurar programas utilizando el entorno "**ddd**". Se sugiere depurar el programa "**psendrec.c**" que figura como anexo.

5 Prueba del ejemplo ejecutándose en varios PC's en paralelo.

Vamos a probar a ejecutar el programa hola con una máquina virtual que incorpore a dos PC's el nuestro y el siguiente de forma circular (pc1 y pc2 en nuestro caso). La idea será arrancar la ejecución del "maestro" en el pc1 y que el resto de los ocho procesos "esclavos", puedan distribuirse entre ambos PC's.

Realizar los pasos siguientes:

- Entrar en el pc1 y situarse en $HOME/p1. Crear el directorio p1 en los otros PC's. Para ello, teclear –para cada PC- comandos como el siguiente: **rsh pc2 mkdir p1**
- Como el ejecutable hola se arrancará en el pc2 de forma remota desde el pc1, debe enviarse dicho ejecutable al pc2. Para ello, ejecutar el comando: "**mpircopy hola**" que lo copiará al $HOME/p1 (directorio de trabajo actual) de todos los PC's salvo desde el que ejecutemos este comando de copia remota. También puede hacerse una copia selectiva a un PC (sea pc2): **mpircopy hola pc2**
- Editar el fichero "holaf.txt" y modificarlo para que tenga el aspecto siguiente:

```
pc1  4  /usuarios/alumnosArqava/arqava01/p1/hola
pc2  4  /usuarios/alumnosArqava/arqava01/p1/hola
```

donde indicamos el deseo de arrancar 4 procesos en el pc1 y otros 4 en el pc2.

- Ahora, ejecutar el programa "hola" tecleando:

  mpirun –p4pg holaf.txt hola

  Nos saldrá algo parecido a lo siguiente:

```
Maestro ejecutandose en pc1
Del proceso 5: Hola, desde pc2
Del proceso 2: Hola, desde pc1
Del proceso 3: Hola, desde pc1
Del proceso 4: Hola, desde pc1
Del proceso 1: Hola, desde pc1
Del proceso 6: Hola, desde pc2
Del proceso 7: Hola, desde pc2
Del proceso 8: Hola, desde pc2
Tiempo del Proceso[1] = 6:0 (seg:mseg)
Tiempo del Proceso[2] = 7:1 (seg:mseg)
Tiempo del Proceso[3] = 8:2 (seg:mseg)
Tiempo del Proceso[4] = 9:1 (seg:mseg)
Tiempo del Proceso[5] = 9:5 (seg:mseg)
Tiempo del Proceso[6] = 9:7 (seg:mseg)
Tiempo del Proceso[7] = 12:432 (seg:mseg)
Tiempo del Proceso[8] = 13:533 (seg:mseg)
P4 procgroup file is holaf.txt.
```

  donde puede observarse cómo se han distribuido los procesos entre las dos máquinas.

- Volver a repetir la prueba utilizando la opción -np tecleando: mpirun -np 9 hola

  Observar la distribución de procesos en PC's y comentarlo con el profesor.

6 Medición de tiempos.

En esta sección vamos a medir lo siguiente:

1. Tiempo de comunicación entre dos procesos que se ejecutan en el mismo PC a través de envío y recepción de mensajes.
2. Tiempo de comunicación entre dos procesos que se ejecutan cada uno en un PC a través del envío y recepción de mensajes.
3. Tiempo de una operación simple.
4. Tiempo de comunicación entre dos procesos que se ejecutan en el mismo PC a través de una variable común y mecanismo de semáforos.

Para tener una aproximación a los tiempos 1 y 2, usaremos el comando "**ping**". Supuesto que estamos en el "pc1" y usaremos "pc2" como PC adicional, podemos teclear lo siguiente:

- ping –c 20 pc1

  La primera comunicación puede ser significativamente mayor que las siguientes, así que para rellenar la Tabla-1 que figura al final de esta sección, puede descartarse y anotar la media del resto de comunicaciones o el valor más frecuente. Observar que el comando ping informa del valor mínimo, la media y el máximo.

- ping –c 20 pc2

  Anotar los resultados del ping entre dos PC's en la Tabla-1

Ahora debemos comprobar lo que nos cuesta esta misma comunicación desde el entorno MPI. Para ello, lo primero es completar el programa "**pingpong**" ejemplo que se muestra en los ANEXOS, programando la parte del envío y recepción que no se suministra escrita.

La forma de evaluar el tiempo de comunicación, es enviar un mensaje a otro proceso y esperar por su respuesta. El tiempo de enviar un mensaje será la mitad del tiempo total medido entre la ida y la vuelta.

A continuación:

- Asegurarse de que estamos situados en $HOME/p1.
- Compilar el programa (*make pingpong*) hasta que no tenga errores de compilación. Tal y como se suministra el programa pingpong de partida, el mensaje de comunicación entre "ping" y "pong" es de un único entero.
- Ejecutar el pingpong en un único PC (*mpirun –p4pg pingpongf.txt pingpong*) y rellenar el apartado correspondiente de la Tabla-1. Observar que el contenido del fichero pingpongf.txt es parecido al siguiente:

  pc1 1 /usuarios/alumnosArqava/**arqava01**/p1/pingpong

- Adecuar el fichero pingpongf.txt para la prueba de comunicación entre dos PC's, de tal forma que ahora su contenido sea algo parecido a:

  pc1 0 /usuarios/alumnosArqava/**arqava01**/p1/pingpong
  pc2 1 /usuarios/alumnosArqava/**arqava01**/p1/pingpong

- Copiarse el ejecutable "pingpong" al otro PC (pc2 en nuestro caso).
- Ejecutar y tomar tiempos.
- Volver a repetir el proceso tras editar y compilar el programa pingpong.c con tamaños de vector de 1.000 y 100.000 enteros, y anotar los tiempos en la Tabla-1.
- Como curiosidad, con el ejecutable preparado para el envío recepción de 100.000 enteros, editar el fichero pingpongf.txt para que tenga el aspecto siguiente:

  pc1 0 /usuarios/alumnosArqava/**arqava01**/p1/pingpong
  pc1 1 /usuarios/alumnosArqava/**arqava01**/p1/pingpong

- Ejecutar esta prueba (pingpong dentro del mismo PC) y observar cómo varían los tiempos respecto de la prueba realizada anteriormente. Preguntarse a qué puede ser debido.
- Una vez rellenada la parte de la Tabla-1 correspondiente al programa "pingpong", comparar con los resultados obtenidos con el comando "ping" y con los resultados obtenidos por el resto de compañeros de prácticas.

Ahora vamos a medir el tiempo de ejecución de una operación simple. Para ello, vamos a utilizar el programa "**costeop**" que se suministra totalmente escrito y cuyo código puede consultarse en los ANEXOS.

Este tipo de mediciones de tiempo de ejecución de operaciones simples puede variar mucho.

Para medir estos tiempos:

- Compilar (*make costeop*) el programa.
- Ejecutarlo cuatro veces simultáneas y anotar el menor de los tiempos. Dado que este programa no utiliza el entorno MPI, basta con teclear el nombre del ejecutable para forzar su ejecución: **./costeop & ./costeop & ./costeop & ./costeop**

Finalmente, para medir el coste de comunicación a través de memoria común, utilizaremos el modelo de *threads* y un programa "**costemc**" tipo productor-consumidor sincronizado mediante semáforos (conceptos vistos en Programación II y Sistemas Operativos I). Este programa también se da completamente escrito y puede consultarse en los ANEXOS.

Para realizar la medición:

- Compilar (*make costemc*) el programa.
- Ejecutarlo cuatro veces simultáneas y anotar el menor de los tiempos. Dado que este programa tampoco utiliza el entorno MPI, basta con teclear el nombre del ejecutable para forzar su ejecución.

Tabla-1. Tiempos en microsegundos

	En mismo PC	Entre dos PC's	
ping			
pingpong 1 int			
pingpong 1.000 int			Calcular Mbps =>
pingpong 100.000 int			Calcular Mbps =>
costeop			**Nota**: Un entero "**int**" ocupa 4 bytes
costemc			

Expresar, normalizado en base al "*costeop*" de una suma condicional, los tiempos siguientes:

costeop	**costemc**	**pingpong 1 int mismo PC**	**pingpong 1 int entre dos PC's**
1			

7 ANEXOS

7.1 Fichero “Makefile”

```
# PCM. Arquitecturas Avanzadas Curso 06/07 EUI           19/09/06  #
#                                                                  #
# Makefile para el desarrollo de las pruebas de los programas      #
#          relacionados con la practica 1 de MPI.                  #

CC          = /usr/local/mpich-1.2.7/bin/mpicc

todo: helloWorld hola pingpong psendrec costeop costemc

helloWorld: helloWorld.c
      $(CC) -Wall -ggdb helloWorld.c -o helloWorld

hola: hola.c
      $(CC) -Wall -ggdb hola.c -o hola

pingpong: pingpong.c
      $(CC) -Wall -ggdb pingpong.c -o pingpong

psendrec: psendrec.c
      $(CC) -Wall -ggdb psendrec.c -o psendrec

costeop: costeop.c
      gcc -Wall costeop.c -o costeop

costemc: costemc.c
      gcc -Wall costemc.c -o costemc -lpthread

borrar:
      rm *.o      helloWorld hola pingpong psendrec costeop costemc
```

7.2 Programa “helloWorld.c”

```
//------------------------------------------------------------------+
// PCM. Arquitecturas Avanzadas Curso 06/07 EUI           19/09/06  |
//                                                                  |
// helloWorld.c: Primer ejemplo de prueba del entorno MPI.          |
//------------------------------------------------------------------+
#include <stdio.h>

#include "mpi.h"

//------------------------------------------------------------------------
int main(int argc, char *argv[]) {
  int yo, numProcesos;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &yo);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcesos);
  if (yo == 0) printf ("Se han creado %d procesos\n", numProcesos);
  printf ("Soy el proceso %d\n", yo);
  MPI_Finalize();
  return 0;
}
```

7.3 Programa "hola.c"

```
//------------------------------------------------------------------+
// PCM. Arquitecturas Avanzadas Curso 06/07 EUI           19/09/06  |
//                                                                  |
// hola.c: Primer ejemplo de prueba del entorno MPI.                |
//------------------------------------------------------------------+
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>

#include "mpi.h"

#define LONG_BUFFER  100

//--------------------------------------------------------------------
void esclavo(int yo) {
  char buffer[LONG_BUFFER];
  struct timeval t0, tf, t;
  long int tiempo[2];         // Segundos y microsegundos

  gettimeofday (&t0, NULL);
  strcpy(buffer, "Hola, desde ");
  gethostname(buffer + strlen(buffer), LONG_BUFFER);
  MPI_Send (buffer, LONG_BUFFER, MPI_BYTE, 0, 1, MPI_COMM_WORLD);
  sleep (yo+5);
  gettimeofday (&tf, NULL);
  timersub(&tf, &t0, &t);
  tiempo[0] = t.tv_sec;
  tiempo[1] = t.tv_usec;
  MPI_Send (tiempo, 2, MPI_LONG, 0, 1, MPI_COMM_WORLD);
}

//--------------------------------------------------------------------
void maestro(int numEsclavos) {
  int i;
  char buffer[LONG_BUFFER];
  long int tiempo[2];         // Segundos y microsegundos
  MPI_Status estado;

  gethostname(buffer, LONG_BUFFER);
  printf ("Maestro ejecutandose en %s\n", buffer);
  for (i=0; i<numEsclavos; i++) {
    MPI_Recv(buffer, LONG_BUFFER, MPI_BYTE, MPI_ANY_SOURCE, 1,
                     MPI_COMM_WORLD, &estado);
    printf("Del proceso %d: %s\n", estado.MPI_SOURCE, buffer);
  }

  for (i=0; i<numEsclavos; i++) {
    MPI_Recv(tiempo, 2, MPI_LONG, i+1, 1, MPI_COMM_WORLD, &estado);
    printf("Tiempo del Proceso[%d] = %ld:%ld (seg:mseg)\n",
              i+1, tiempo[0], tiempo[1]/1000);
  }
}
```

```
//------------------------------------------------------------------------
int main(int argc, char *argv[]) {
  int yo, numProcesos;

  setbuf (stdout, NULL); // Sin buffers de escritura
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &yo);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcesos);
  if (yo == 0) maestro(numProcesos-1);
  else         esclavo(yo);
  MPI_Finalize();
  return 0;
}
```

7.4 Programa “pingpong.c”

```
// PCM. Arquitecturas Avanzadas Curso 07/08 EUI           11/10/07   |
//                                                                   |
// pingpong.c: Programa de prueba para evaluar los tiempos de        |
//             latencia de las comunicaciones.                       |

#include <assert.h>
#include <sys/time.h>
#include <stdio.h>

#include "mpi.h"

#define ITERACIONES     200
#define LONG_MSJ          1  // Probar con 1, 1000 y 100000

//--------------------------------------------------------------------
void pong(void) {
  int basura[LONG_MSJ];  // Mensaje a intercambiarse

    //    Rellenar adecuadamente
}

//--------------------------------------------------------------------
void ping(void) {

  int basura[LONG_MSJ];  // Mensaje a intercambiarse
  int i;
  struct timeval t0, t1, tiempo;
  long int microseg = 0;

  for (i=0; i<=ITERACIONES; i++) {
    assert (gettimeofday (&t0, NULL) == 0);

    // Envio y recepcion a "pong"
    //    Rellenar adecuadamente

    assert (gettimeofday (&t1, NULL) == 0);
    timersub(&t1, &t0, &tiempo);
    assert (tiempo.tv_sec == 0);
    printf ("%6ld", tiempo.tv_usec / 2);
    if (((i+1)%15) == 0) printf ("\n");
    // Desprecio la primera comunicacion
    if (i>0) microseg = microseg + tiempo.tv_usec;
  }
  printf ("\nTiempo medio = %ld microseg\n", microseg/(ITERACIONES*2));
}

//--------------------------------------------------------------------
int main( int   argc, char *argv[] ) {
  int yo;

  setbuf (stdout, NULL); // Sin buffers de escritura
  MPI_Init (&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &yo);
  if (yo == 0) ping();
  else         pong();
  MPI_Finalize();
  return 0;
}
```

7.5 Programa “costeop.c”

```
//------------------------------------------------------------------+
// PCM. Arquitecturas Avanzadas Curso 08/09 EUI          21/10/08  |
//                                                                  |
// costeop.c: Prueba de calculo del tiempo de ejecucion de una      |
//               una operacion simple condicional.                  |
//------------------------------------------------------------------+

#include <stdio.h>
#include <sys/time.h>
#include <assert.h>

#define LONG_VECTOR 10000000
#define NUM_VECES   10000000

static int vector[LONG_VECTOR];

//--------------------------------------------------------------------
void computar () {
  int i, j;

  j = 5;
  for (i=0; i < NUM_VECES; i++) {
    // j = (j + 1) % 3333;
    if (j==3333) j = 0;
    else         j++;
  }
  vector[j] = 0;
}

//--------------------------------------------------------------------
int main (int argc, char *argv[])
{
  struct timeval t0, tf, tiempo;
  float  microseg;

  assert (gettimeofday (&t0, NULL) == 0);
  computar();
  assert (gettimeofday (&tf, NULL) == 0);
  timersub (&tf, &t0, &tiempo);
  assert (tiempo.tv_sec == 0);
  microseg = (float) (tiempo.tv_usec) / NUM_VECES;
  printf ("Tiempo de operacion basica: %7.5f (microseg)\n", microseg);
  return 0;
}
```

7.6 Programa “costemc.c”

```
//------------------------------------------------------------------+
// PCM. Arquitecturas Avanzadas Curso 06/07 EUI          19/09/06  |
//                                                                  |
// costemc.c: Prueba del coste de comunicacion entre dos threads a  |
//            traves de una posicion de memoria (uno escribe y el   |
//            otro lee.                                             |
//------------------------------------------------------------------+
#include <assert.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <sys/time.h>
```

```
#define NUM_VECES   100000

// Variables globales para que ambos procesos conozcan
// el identificador de thread del otro proceso
static pthread_t t_productor, t_consumidor;
static int datoComun = 0;
static long int microseg;
static sem_t sPro, sCon;

//-----------------------------------------------------------------
void *productor(void *basura) {
  struct timeval t0, tf, t;
  int i;

  assert (gettimeofday (&t0, NULL) == 0);
  for (i=0; i<NUM_VECES; i++) {
    assert (sem_wait (&sPro) == 0);
      datoComun++;
    assert (sem_post (&sCon) == 0);
  }
  assert (gettimeofday (&tf, NULL) == 0);
  timersub (&tf, &t0, &t);
  microseg = t.tv_sec * 1000000 + t.tv_usec;
  printf ("Tiempo sincronizacion por memoria comun = %7.3f microseg\n",
            (float) microseg / (float) NUM_VECES);
  pthread_exit (NULL);
}

//-----------------------------------------------------------------
void *consumidor(void *basura) {

  int i;

  for (i=0; i<NUM_VECES; i++) {
    assert (sem_wait (&sCon) == 0);
      assert (datoComun == i+1);
    assert (sem_post (&sPro) == 0);
  }
  pthread_exit (NULL);
}

//-----------------------------------------------------------------
int main (int argc, char *argv[])
{

 assert (sem_init (&sPro, 0, 1) == 0);
 assert (sem_init (&sCon, 0, 0) == 0);
 assert (pthread_create(&t_productor,  NULL, productor,  NULL) == 0);
 assert (pthread_create(&t_consumidor, NULL, consumidor, NULL) == 0);
 assert (pthread_join (t_productor,  NULL) == 0);
 assert (pthread_join (t_consumidor, NULL) == 0);
 return 0;
}
```

7.7 Programa “psendrec.c”

```
//------------------------------------------------------------------+
// PCM. Procesamiento Paralelo  Curso 05/06 EUI            9/02/06  |
//                                                                  |
// psendrec.c: Prueba simple de MPI_Send y MPI_Recv                 |
//------------------------------------------------------------------+

#include <stdio.h>
#include <unistd.h>

#include "mpi.h"

#define N     3
#define VECES 5

//--------------------------------------------------------------------
void esclavo(void) {
  int i, j, tabla[N], n;
  MPI_Status estado;

  sleep(2);
  for (i=0; i<VECES; i++) {
    MPI_Recv (tabla, N, MPI_INT, 0, 1, MPI_COMM_WORLD, &estado);
    MPI_Get_count (&estado, MPI_INT, &n);
    printf ("E: recibe => ");
    for (j=0; j<N; j++) printf("%d ", tabla[j]);
    printf (" de tid = %d eti = %d elementos = %d \n",
            estado.MPI_SOURCE, estado.MPI_TAG, n);
  }
}

//--------------------------------------------------------------------
void maestro (void) {
  int i, j, vector[N];

  for (i=0; i<VECES; i++) {
    printf ("M: envia => ");
    for (j=0; j<N; j++) {
      vector[j] = i*N+j;
      printf("%d ", vector[j]);
    }
    printf ("\n");
    MPI_Send (vector, N, MPI_INT, 1, 1, MPI_COMM_WORLD);
  }
}

//--------------------------------------------------------------------
int main( int   argc, char *argv[] ) {
  int yo;

  setbuf(stdout, NULL); // Sin buffers en escritura
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &yo);
  if (yo ==  0) maestro();
  else          esclavo();
  MPI_Finalize();
  return 0;
}
```