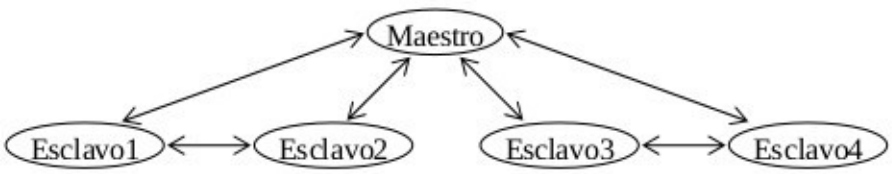
Práctica 1
Primera experiencia con MPI y MPE

1 Visión global de MPI

MPI (Message Passing Interface) es un estándar para programación paralela basada en el envío de mensajes en arquitecturas con memoria distribuida. El primer estándar se publicó en 1994 y, posteriormente, se generó una segunda versión "MPI-2" publicada en 1997. La razón principal para desarrollar MPI fue que cada fabricante de MPP (Massively Parallel Processor) estaba creando su propia biblioteca de computación paralela y distribuida basada en paso de mensajes, de forma que no sería posible desarrollar aplicaciones paralelas portables entre sistemas. A partir de estas definiciones existen implementaciones tanto comerciales como de libre distribución (nosotros utilizaremos MPICH). MPI es, junto con PVM, "Parallel Virtual Machine", uno de los paquetes de software más populares en el entorno de aplicaciones basadas en paso de mensajes. Entre las principales características de MPI destacan:

- un amplio conjunto de rutinas de comunicación punto-a-punto,
- un amplio conjunto de rutinas de comunicación entre grupos de procesos,
- un contexto de comunicación para el diseño de bibliotecas paralelas,
- la posibilidad de especificar diferentes topologías de comunicación,
- la posibilidad de crear tipos de datos derivados para enviar mensajes que contengan datos no contiguos en memoria,
- existen varias implementaciones de calidad de MPI disponibles (LAM, MPICH, CHIMP, OpenMPI),
- existe una implementación de MPI para C++, C, Fortran e incluso para Java,
- en MPI se puede hacer uso de comunicación asíncrona,
- MPI maneja más eficientemente el paso de mensajes,
- usando MPI se pueden desarrollar aplicaciones más eficientes en MPP y clusters,
- MPI es totalmente portable,
- existe una especificación formal de MPI,
- MPI es un estándar (sus características y comportamiento se acordaron en un foro abierto).

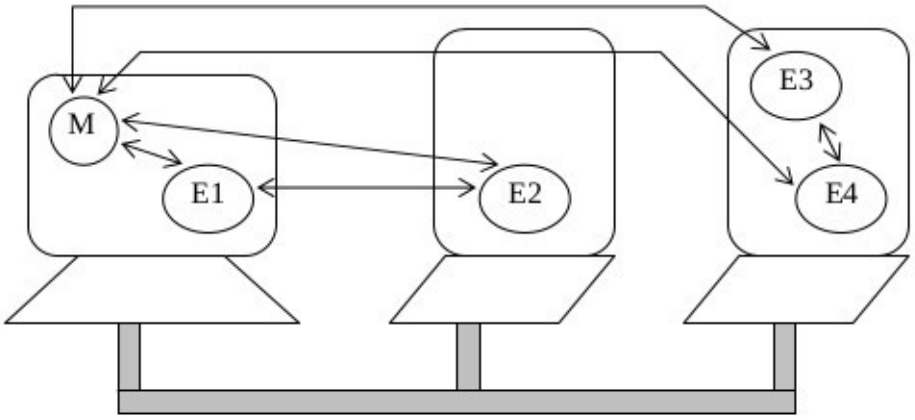
Una aplicación paralela típica sería aquella en la que un proceso maestro crea varios procesos esclavos para realizar, cada uno de ellos, una parte del trabajo global. Un ejemplo de este modelo se presenta en la siguiente figura:



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

Dependiendo del soporte del sistema de operación, esta aplicación podría ejecutarse en un único procesador intercalando el uso del CPU entre los cinco procesos y la comunicación pudiera ser a través del envío de mensajes.

Si disponemos de más de una máquina física interconectadas 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 donde se ejecuta la aplicación paralela de tal forma que los procesos se distribuyen entre los distintos CPU's disponibles. Por ejemplo, si se dispone de tres PCs, la aplicación antes descrita 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 resaltar 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.

2 Primitivas básicas de MPI para crear procesos y comunicación

Consultar la referencia "www-unix.mcs.anl.gov/mpi/www/" para una descripción completa de las funciones de MPI.

Todas las funciones de MPI devuelven un valor entero que corresponde al código de error.

Control de procesos

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Devuelve en “size” el número de procesos asociados al grupo “comm”. 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 comm, int *rank)
```

Esta función devuelve en “rank” el rango del proceso que la invoca dentro del grupo “comm” indicado. Los identificadores van del cero en adelante (hasta el número de elementos de dicho grupo).

```
int MPI_Finalize ( )
```

Le indica al entorno MPI que el proceso que la invoca 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)
```

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

A continuación se presenta un ejemplo que utiliza las funciones antes vistas.

```
int main(int argc, char *argv[]) {
   int me, numProcess;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &me);
   MPI_Comm_size(MPI_COMM_WORLD, &numProcess);
   if (me == 0) printf ("Se han creado %d procesos\n", numProcess);
      printf ("Soy el proceso %d\n", me);
   MPI_Finalize();
   return 0;
}
```

Si compilamos y ejecutamos el programa anterior con los siguientes comandos

```
[car]> /usr/mpich/bin/mpicc HelloWorld.c -o HelloWorld

[car]> /usr/mpich/bin/mpirun -machinefile mis_maquinas -np 3 HelloWorld
```

donde –np 3 indica que se arranquen tres procesos que ejecutarán el programa HelloWorld en tres de los nodos especificados en el archivo mis_maquinas. La salida podría ser la siguiente:

```
Soy el proceso 2
```

```
Soy el proceso 1
Se han creado 3 procesos
Soy el proceso 0
```

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 arreglo de datos del mismo tipo.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)
```

El significado de los parámetros es el siguiente:

`buf`	Dirección inicial del arreglo o buffer a enviar;
`count`	Número de elementos a enviar (igual o mayor que cero);
`datatype`	Tipo de los elementos del arreglo a enviar;
`dest`	Rango del proceso destinatario;
`tag`	Etiqueta del mensaje. Debe estar en el rango 0..MPI_TAG_UB;
`comm`	Grupo de comunicación. En principio MPI_COMM_WORLD.

Los posibles valores de los tipos de datos que pueden ser enviados o recibidos son:

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.

El envío de mensajes puede bloquear al proceso que hace el envío hasta que el destinatario haya recibido el mensaje. El bloqueo dependerá de la posibilidad de almacenar temporalmente en el buffer en el caso de que el receptor no esté esperando el mensaje, y es dependiente de la implementación. En este punto hay que ser muy cuidadosos pues si los mensajes son grandes y dependiendo del patrón que se use en la comunicación puede ocurrir un deadlock entre los procesos.

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Status *status)
```

El proceso que invoca MPI_Recv se bloquea hasta recibir el mensaje indicado.

El significado de los parámetros es el siguiente:

`buf` Dirección del arreglo donde se dejará el mensaje recibido;

`count` Número máximo de elementos en el buffer de recepción. El número de elementos realmente recibidos puede conocerse con MPI_Get_count;

`datatype` Tipo de los datos del arreglo (los mismos que en MPI_Send);

`source` Rango del proceso remitente. También puede indicarse MPI_ANY_SOURCE, en cuyo caso se indica que se recibirá un mensaje de cualquier remitente;

`tag` Etiqueta del mensaje. También puede indicarse MPI_ANY_TAG, en cuyo caso se indica que se recibirá un mensaje con cualquier etiqueta;

`comm` Grupo de comunicación. En principio MPI_COMM_WORLD;

`status` Resultado de la recepción. Se trata de una estructura que tiene, entre otros, los campos siguientes:

MPI_SOURCE	indica quién envió el mensaje. Útil en combinación con MPI_ANY_SOURCE
MPI_TAG	indica la etiqueta del mensaje recibido. Útil en combinación con MPI_ANY_TAG

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

Devuelve en “counts” el número de elementos del tipo *datatype* recibidos por la operación de recepción que devolvió el status indicado como primer parámetro.

```
int MPI_Bcast (void *buf, int count, MPI_Datatype datatype, int source,
MPI_Comm comm)
```

Esta primitiva envía un mensaje desde el proceso source al resto de los procesos que pertenecen al grupo *comm*.

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

Los parámetros "buf", "count" y "datatype" identifican el vector que se va a enviar en el caso del proceso "source", y el vector donde se recibirá la información en el caso de los receptores.

3 Programa de prueba

HelloWorld.c

- Revisar y ejecutar HelloWorld indicando el número de procesos a usar y en qué nodos serán ejecutados

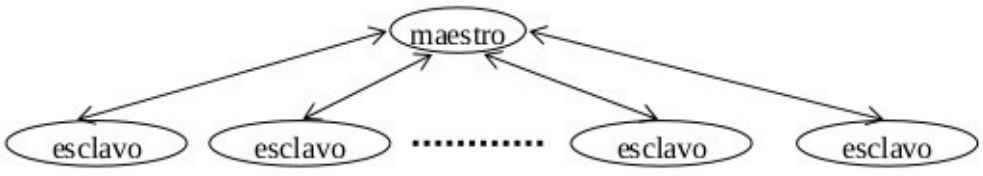
```
$> mpicc HelloWorld.c -o HelloWorld
$> mpirun -machinefile mis_maquinas -np 3 HelloWorld
```

- Posible salida

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

Hola.c

Vamos ejecutar un programa de ejemplo que consiste en un proceso "maestro" que se comunica con n-1 procesos "esclavos". Gráficamente:



El proceso "maestro" indica dónde se ejecuta y espera un mensaje de cada "esclavo". En dicho mensaje cada "esclavo" le informa al maestro en qué máquina está ejecutando. Después recibe de cada uno de ellos otro mensaje con el tiempo que han tardado en

ejecutar. De todo ello informará por la salida estándar.

Cada proceso “esclavo” lo que hará básicamente es una espera mediante “sleep”, específicamente “sleep (me + 5)”, de forma que el primer proceso espera sólo 6 segundos, el siguiente 7 segundos y así sucesivamente. En el archivo hola.c se encuentra tanto el código del maestro como el de los esclavos.

Ejercicio:

- Revisar y entender hola.c
- Compilar y ejecutar
- Posible salida

```
mpirun -np 4 -machinefile mis_maquinas a.out
Maestro ejecutandose en car13.labf.usb.ve
Del proceso 2: Hola, desde car07.labf.usb.ve
Del proceso 1: Hola, desde car05.labf.usb.ve
Del proceso 3: Hola, desde car02.labf.usb.ve
Tiempo del Proceso[1] = 6:6 (seg:mseg)
Tiempo del Proceso[2] = 7:7 (seg:mseg)
Tiempo del Proceso[3] = 8:9 (seg:mseg)
```

4. **Análisis del Comportamiento de un programa**

Escribir un programa paralelo es más complejo que escribir programas seriales. Para desarrollar programas paralelos eficientes y escalables es necesario entender el comportamiento del mismo. Para lograr este objetivo se puede usar una técnica que consiste en el registro de la ocurrencia de una serie de eventos (archivo de trazas) y su posterior visualización. Los eventos registrados tienen asociados el instante de tiempo de ocurrencia y algún dato adicional, y a partir de estas trazas se puede realizar un análisis post mortem del comportamiento del programa.

La librería MPE (MultiProcessing Environment) provee una serie de facilidades para recoger información del comportamiento de programas en MPI para realizar análisis post mortem. Fue desarrollado para la implementación MPICH de MPI. El enlace de esta librería con programas en MPI generará un logfile cuando el mismo es ejecutado.

cpilog.c

- Revisar y entender cpilog.c. Este archivo incluye una serie de llamadas a funciones de la librería MPE, entender cada una de estas llamadas.
- Compilar y ejecutar.
- Ejecutar jumpshot para visualizar la información obtenida a partir de la traza de eventos generada por MPE.

5 ANEXOS

5.1 Programa “helloWorld.c”

```
//------------------------------------------------------------------+
// Introducción al paralelismo Abril-Julio 2016                     |
//                                                                  |
// helloWorld.c: Primer ejemplo de prueba del entorno MPI.          |
//------------------------------------------------------------------+
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[]) {
     int me, numProcess;
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &me);
     MPI_Comm_size(MPI_COMM_WORLD, &numProcess);
     if (me == 0) printf ("Se han creado %d procesos\n", numProcess);
     printf ("Soy el proceso %d\n", me);
     MPI_Finalize();
     return 0;
}
```

5.2 Programa “hola.c”

```
//--------------------------------------------------------------------+
// Introducción al paralelismo Abr - Julio 2016                       |
//                                                                    |
// hola.c: 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 me) {
      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 (me+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 rank, numProcess;

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

5.3 Programa “cpilog.c”

Cálculo de pi en paralelo con registro de eventos para su posterior visualización con jumpshot.

```
#include "mpi.h"
#include "mpe.h"
#include <math.h>
#include <stdio.h>

double f( double a );
double f(double a)
{
      return (4.0 / (1.0 + a*a));
}

int main( int argc, char *argv[] )
{
      int n, myid, numprocs, i, j;
      double PI25DT = 3.14159265358979323846264 3;
      double mypi, pi, h, sum, x;
      double startwtime=0.0, endwtime;
      int namelen;
      int event1a, event1b, event2a, event2b,
          event3a, event3b, event4a, event4b;
      char processor_name[MPI_MAX_PROCESSOR_NAME];

      MPI_Init(&argc,&argv);
      MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
      MPI_Comm_rank(MPI_COMM_WORLD,&myid);

      MPI_Get_processor_name(processor_name,&namelen);
      fprintf(stderr,"Process %d running on %s\n", myid, processor_name);

      /*
            MPE_Init_log() & MPE_Finish_log() are NOT needed when
            liblmpi is linked with this program. In that case,
            MPI_Init() would have called MPE_Init_log() already.
      */

      MPE_Init_log();
      /* Get event ID from MPE, user should NOT assign event ID  */
      event1a = MPE_Log_get_event_number();
      event1b = MPE_Log_get_event_number();
      event2a = MPE_Log_get_event_number();
      event2b = MPE_Log_get_event_number();
      event3a = MPE_Log_get_event_number();
      event3b = MPE_Log_get_event_number();
      event4a = MPE_Log_get_event_number();
      event4b = MPE_Log_get_event_number();

      if (myid == 0) {
            MPE_Describe_state(event1a, event1b, "Broadcast","red");
            MPE_Describe_state(event2a, event2b, "Compute", "blue");
            MPE_Describe_state(event3a, event3b, "Reduce", "green");
            MPE_Describe_state(event4a, event4b, "Sync", "orange");
      }
```

```
	if (myid == 0)
	{
		n = 1000000;
		startwtime = MPI_Wtime();
	}
	MPI_Barrier(MPI_COMM_WORLD);

	MPE_Start_log();
	for (j = 0; j < 5; j++)
	{
		MPE_Log_event(event1a, 0, "start broadcast");
		MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
		MPE_Log_event(event1b, 0, "end broadcast");

		MPE_Log_event(event4a,0,"Start Sync");
		MPI_Barrier(MPI_COMM_WORLD);
		MPE_Log_event(event4b,0,"End Sync");

		MPE_Log_event(event2a, 0, "start compute");
		h = 1.0 / (double) n;
		sum = 0.0;
		for (i = myid + 1; i <= n; i += numprocs)
		{
			x = h * ((double)i - 0.5);
			sum += f(x);
		}
		mypi = h * sum;
		MPE_Log_event(event2b, 0, "end compute");

		MPE_Log_event(event3a, 0, "start reduce");
		MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
		MPE_Log_event(event3b, 0, "end reduce");
	}
	MPE_Finish_log("cpilog");
	if (myid == 0)
	{
		endwtime = MPI_Wtime();
		printf("pi is approximately %.16f, Error is %.16f\n",
			pi, fabs(pi - PI25DT));
		printf("wall clock time = %f\n", endwtime-startwtime);
	}
	MPI_Finalize();
	return(0);
}
```

5.4 Programa “psendrec.c”

```
#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;
}
```