P2S1

# Práctica 2:
# Programación con MPI. Sesión 1.

(Actualizado el 04/04/2017)

Tiempo de trabajo en casa (previsto): 3:00h
Tiempo de laboratorio: 2:30h
Evaluación de la práctica: 30 m.

## 1. Introducción

El modelo de programación basado en MPI se ha convertido en uno de los estándares actuales en la programación paralela mediante paso de mensajes. En este modelo de programación cada proceso tiene su espacio de direcciones propio, por lo que es el adecuado para los sistemas multicomputador. Las comunicaciones y sincronización entre procesos se llevan a cabo mediante comunicaciones entre los procesos, siguiendo el modelo de ejecución CSP de Hoare. La programación con MPI sigue una estructura SPMD, en el que cada proceso ejecuta una parte del código del programa en función de su identificador y posee funciones que permiten realizar comunicaciones colectivas mediante una única llamada a las funciones en el código. Así, MPI define un entorno de programación único y portable entre plataformas.

Esta Práctica consta de 3 sesiones de laboratorio, una inicial en la que revisaremos los conceptos vistos en clase de forma guiada y supervisada por el profesor, y dos sesiones en las que se trabajará la paralelización del problema de estudio elegido por cada grupo. Por tanto, en esta primera sesión se pretende dar a conocer las diferentes utilidades que proporciona MPI para el diseño y ejecución de programas paralelos en multicomputadores y computadores multinúcleo.

Como ya hemos comentado en la práctica anterior, en las sesiones de laboratorio de la asignatura se tratará de afianzar los conocimientos vistos en las clases teóricas y se continuará el desarrollo de las técnicas básicas para la toma de datos y su representación en una tabla o gráfica. Para ello, antes de la sesión de laboratorio se pedirá a los miembros de cada grupo de trabajo que lean la documentación relacionada con el fundamento teórico del trabajo a desarrollar y las herramientas a utilizar. La lectura previa de estos documentos permitirá utilizar las aplicaciones y el equipamiento del laboratorio, y responder a las preguntas acerca de su funcionalidad y uso.

En las sesiones de laboratorio es importante establecer una dinámica de trabajo en grupo que permita sacar el máximo provecho de tu estancia en el laboratorio. Para que ello sea posible, se requiere que seas puntal y que prestes atención a las indicaciones del profesor.

### Objetivos

El objetivo de esta práctica es aprender a trabajar con MPI, estudiando el impacto de las distintas opciones de compilación en la ejecución de los programas. También descubriremos la importancia que tiene el conocimiento profundo de la arquitectura de los computadores con los que trabajamos y su red de interconexión en el aprovechamiento de sus recursos.

Al finalizar esta práctica, el estudiante será capaz de:

- Compilar y ejecutar programas con MPI

- Usar las funciones de envío y recepción de mensajes punto a punto (MPI_Send, MPI_Recv, etc.).
- Usar las funciones colectivas (MPI_Bcast, MPI_Reduce, MPI_Scatter, MPI_Gather, etc.)
- Calcular el tiempo de ejecución de un programa con MPI utilizando las funciones: CLOCK, gettimeofday y MPI_Wtime (recomendado).
- Utilizar las diversas opciones de ejecución de los programas con MPI (http://www.open-mpi.org/doc/v1.6/man1/mpirun.1.php).
- Hacer el estudio de escalabilidad de un problema.

### Recursos

Para el desarrollo de estas sesiones de laboratorio se dispondrá de un computador personal PC con el que se conectará a un servidor (**boe.uv.es**) con Sistema Operativo Linux Centos 6.5.

El servidor **boe** posee dos procesadores multinúcleo Intel Xeon E5-2620V2 a 2,1 GHz con 6 núcleos cada uno (un total de 12 CPU con hyper-threading), con tecnología de 22 nm, 80W de disipación de potencia y una memoria principal DDR3/1600 de 64 GB. Cada núcleo posee tres niveles de Cache: en el nivel L2 hay 0.25 MB por núcleo, y el L3 de cache es unificada de 15MB (2,5 MB por núcleo).

El servidor boe está conectado mediante una red interna GEthernet a 8 nodos de cálculo llamados “compute-0-0” - “compute-0-7”. Cada nodo posee un procesador Intel Xeon E5-2620V2 como los del servidor, pero con una memoria principal de 32GB.

El compilador de C/C++ instalado es el gcc/g++ versión 4.4.7, y el MPI instalado es la versión OpenMPI 1.6.2.

### Aprendizaje y evaluación del laboratorio.

El aprendizaje y la evaluación de la práctica comenzarán antes de asistir al laboratorio. Así, el estudiante deberá reunirse con sus compañeros de grupo para preparar cada sesión. El trabajo previo a la práctica consistirá en revisar los conceptos de teoría con los que está relacionada la práctica y comenzar a realizar las actividades propuestas para preguntar las dudas surgidas en la propia sesión o en tutorías. Todo ello nos preparará para obtener el máximo provecho de la sesión de laboratorio.

Durante el desarrollo de la práctica se evaluará el trabajo desarrollado y los logros obtenidos por el grupo en la sesión presencial en el laboratorio. Habrá diversos apartados, formado por actividades de aprendizaje. A medida que se vaya completando cada actividad propuesta, el estudiante debe llamar al profesor para que compruebe qué ha hecho hasta ese momento. El profesor asignará entonces la puntuación correspondiente a la actividad que se haya completado según su grado de exactitud y las respuestas que el grupo de a las preguntas del profesor.

Al final de cada sesión o grupo de sesiones que constituyan una práctica se realizará una evaluación individual relacionada con las actividades y objetivos de ésta.

## 2. Trabajo previo al Laboratorio – Pre-Lab

Para poder sacar el máximo partido a la sesión de laboratorio es muy importante que leas la práctica y realices parte de las actividades que se proponen, dedicando el tiempo que se indica. Para que este tiempo no supere el tiempo máximo estimado, los miembros del equipo de trabajo

deberéis distribuiros las actividades propuestas y realizar posteriormente una puesta en común. Una mala distribución de las mismas, o que totas las realice un solo miembro, supondrá no poder realizarlas todas o una dedicación de tiempo que estaréis restando de otras actividades.

## 3. Trabajo en el Laboratorio – In-Lab

En esta sesión se aprenderá a usar los elementos básicos de MPI sobre un conjunto de programas ejemplo. Así, en los siguientes puntos se propondrán un conjunto de actividades a realizar seguidas por cuestiones a resolver o acciones que deberás experimentar.

### 3.1 Inicio: Hola

Vamos a iniciar la práctica con un programa típico (”Hola mundo”) se ejecutan de forma paralela. A continuación, se muestra el código del programa:

```
/* Ejemplo Hola */
#include <stdio.h>
#include <mpi.h>
#include <unistd.h>

int main (argc, argv)
     int argc;
     char *argv[];
{
  int rank, size;
  char hostname[256];
    MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD,&rank);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  gethostname(hostname,255);
  printf( "Hola desde el proceso %d de %d en el nodo %s \n", rank,
  size, hostname);
  MPI_Finalize();
  return 0;
}
```

Como puedes ver los programas con MPI deben incluir

```
#include <mpi.h>
```

para hacer uso de las librerías de MPI.

Para compilar el código en C utilizaremos la sentencia:

```
> mpicc -o hola hola.c
```

Y para ejecutar el programa simplemente pondremos:

```
>mpirun –np p ./hola
```

La salida del programa dependerá del número de procesos “*p*” con los que se ha ejecutado. Por defecto los procesos se ejecutan en el host (boe), pero para asegurarse de las máquinas que vamos a utilizar es bueno que definamos un fichero propio (*maquinas*) que contenga la relación de los nodos que puedo utilizar. En este primer caso contendrá:

```
boe.uv.es slots=12
```

(donde slots indica el nº de procesos que pueden ejecutarse potencialmente en un nodo)

Y en la ejecución utilizaremos el flag *–hostfile* o *-machinefile*:

```
>mpirun –np p –hostfile maquinas ./hola
```

Cuando utilicemos el resto de nodos de computación los añadiremos a *maquinas*. Hay otras opciones que permiten realizar diferentes asignaciones de procesos a nodos, que puedes encontrar en http://www.open-mpi.org/doc/v1.6/man1/mpirun.1.php.

A continuación, realiza las siguientes pruebas y responde a las siguientes cuestiones:

1) Comenta en el código ejemplo qué hace cada una de las funciones de MPI invocadas.
2) Compila el programa propuesto y ejecútalo con 12 procesos. Asegura que todos los procesos se ejecutan en el host.
3) Modifica el código ejemplo para que dependiendo del valor del identificador de cada proceso indique si es par o impar. Ejemplo: para 4 procesos tendríamos una salida del tipo:

   > Hola desde un proceso par de 4

   > Hola desde un proceso impar de 4

   > Hola desde un proceso impar de 4

   > Hola desde un proceso par de 4

4) Añade el resto de màquinas del clúster al fichero maquinas.
5) ¿Qué ocurre si se ejecuta con la opción –npernode 1?
6) ¿Qué ocurre si la ejecución se hace con el flag -nolocal? ¿Y si eliminamos del fichero maquinas la entrada de boe?

### 3.2 Envío y recepción

Para enviar y recibir mensajes entre procesos se utilizan las funciones MPI_Send y MPI_Recv. Con estas funciones es posible incluso que un proceso se enviara un mensaje a sí mismo. A continuación, se muestra el código que utilizaremos como ejemplo:

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int rank, contador;
    MPI_Status estado;
    MPI_Init(&argc, &argv); // Inicio la comunicacion de procesos
    MPI_Comm_rank(MPI_COMM_WORLD,&rank); // Obtiene valor id propio
    //Envia y recibe mensajes
    MPI_Send(&rank //referencia al elemento a enviar
            ,1 // tamaño del vector a enviar
            ,MPI_INT // Tipo de dato que se envia
            ,rank // id del proceso destino
            ,0 //etiqueta
            ,MPI_COMM_WORLD); //Comunicador por el que se manda
    MPI_Recv(&contador // Referencia donde se almacena lo recibido
            ,1 // tamaño del vector a recibir
            ,MPI_INT // Tipo de dato que recibe
            ,rank // id del proceso origen del que se recibe
            ,0 // etiqueta
```

```
            ,MPI_COMM_WORLD // Comunicador por el que se recibe
            ,&estado); // estructura informativa del estatus
    printf("Soy el proceso %d y he recibido %d\n",rank,contador);
    MPI_Finalize();
    return 0;
}
```

Utilizando este código como base queremos hacer un programa paralelo que encadene el envío y recepción de un mensaje, que en nuestro caso será el identificador del proceso que envía.

Los mensajes se enviarán de forma encadenada, de manera que el primero enviará su identificador al segundo, el segundo recibirá este del primero y lo enviará el suyo al tercero, y así sucesivamente para todos los procesos.

Todo proceso que reciba un mensaje debe imprimirlo de la forma (Nota: en este caso el último proceso no envía ningún mensaje):

*"Soy el proceso x y he recibido m", siendo x el rango del proceso y m el mensaje recibido.*

1) Modifica el código anterior para que tenga la funcionalidad descrita. Compílalo y ejecuta todos los procesos en el host (boe.uv.es)
2) Modifica el código diseñado para que en la recepción de los mensajes no se tenga en cuenta la etiqueta o *tag* de los mensajes. Ejecútalo igual que antes.
3) Modifica el código anterior para que el mensaje dé la vuelta completa y llegue al proceso 0. Elimina todas las impresiones, salvo la del proceso 0, y mide el tiempo en que tarda el mensaje en hacer todo el recorrido, imprimiendo por pantalla este valor para los siguientes casos:
    a. Todos los procesos se ejecutan en el host.
    b. Cada proceso se ejecuta en una máquina, comenzando por el host.

### 3.3 Calculo de π

Después de ver las funciones que envían mensajes entre procesos “punto a punto” vamos a tratar las operaciones colectivas MPI_Bcast y MPI_Reduce.

Como caso de ejemplo utilizaremos el siguiente código que calcula el valor de π mediante la integración de Riemann:

```
#include <math.h>
#include <stdlib.h> // Incluido para el uso de atoi
#include <stdio.h>

int main(int argc, char *argv[])
{
  // Cálculo de PI
  int n;
  printf("Introduce la precisión del cálculo (n > 0): ");
  scanf("%d",&n);
  double PI25D = 3.14159265358979323846264 3;
  double h = 1.0 / (double) n;
  double sum = 0.0;
  for (int i = 0; i < n; i++) {
       double x = h * ((double)i + 0.5);
```

```
        sum += (4.0 / (1.0 + x*x));
    }
    double pi = sum * h;
      printf ("El valor aproximado de PI es: %f, con un error de %f
\n",pi,fabs(pi - PI25D));
    return 0;
}
```

La aproximación de una integral mediante la suma de Riemann permite dividir el trabajo en unidades independientes, siendo un factor de precisión el número de divisiones *n*.

En el programa que se propone realizar, los procesos creados se distribuyen las *n* iteraciones del bucle y acumulan los valores calculados en cada iteración en una variable local a cada proceso. El factor de precisión *n* puede pedirlo el proceso raíz (0) y repartirlo a los otros procesos mediante MPI_Bcast. Cuando mayor es el valor de *n* se supone que mayor precisión va a tener el valor de π calculado.

Después de que cada proceso calcule su parte, se han de reunir todas las partes en el proceso raíz (0) para mostrar el resultado con MPI_Reduce. No olvides lo aprendido en los puntos anteriores.

1) Diseña el programa propuesto y ejecútalo para diferentes precisiones y número de procesadores.
2) Ejecuta los procesos solo en el host y utilizando todos los nodos disponibles con un proceso en cada nodo.

### 3.4 Producto escalar

En este apartado vamos a tratar las funciones colectivas que sirven para distribuir una estructura de datos entre los procesos. En este caso utilizaremos la función MPI_Scatter.

A continuación, se presenta un programa secuencial que calcula el producto escalar de dos vectores. En el código se han introducido diversas formas de medir el tiempo de ejecución (CLOCK, gettimeofday y MPI_Wtime).

```
/* seq_dot.c – calcula un producto escalar de forma secuencial.
 *
 * Input:
 *     n: talla de los vectores
 *  x, y: los vectores
 *
 * Output:
 *     el producto escalar de x por y.
 *  Nota:  en  la  versión  paralela  n  es  múltiplo  del  número  de
procesadores
 */
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>
#include "mpi.h"

#define MAXN 80000000
#define TUNIT 1.0e+6

main(int argc, char* argv[])
```

```
{
  double *x, *y;
  double dot, local_dot;
  int    i, n;
  double start, finish, dt1;
  struct timeval tv1, tv2;
  clock_t tstart,tend;
  double cpu_time_used;

  MPI_Init(&argc, &argv);
if (argc<2){
    printf("Entra el número de elementos de cada vector: \n");
    scanf("%d", &n);
  }
else
    n=atoi(argv[1]);

  // Reserva de espacio para los vectores
    x = (double *) calloc(n, sizeof(double));
    y = (double *) calloc(n, sizeof(double));
  // Inicio de los vectores a 1
    for (i=0; i<n; i++) {x[i] = 1.0; y[i] = 1.0;}

//Toma de tiempos iniciales
    tstart = clock();
    gettimeofday(&tv1, (struct timezone*)0);
    start = MPI_Wtime();

  /******* calcula el product escalar */
  dot = 0.0;
  for (i = 0; i < n; i++)
    dot += x[i] * y[i];
  free(x); free(y);

//Toma de tiempos finales
    tend = clock();
    gettimeofday(&tv2, (struct timezone*)0);
    finish = MPI_Wtime();

  dt1= (tv2.tv_sec - tv1.tv_sec) * 1000000.0 + (tv2.tv_usec -
tv1.tv_usec);
  cpu_time_used = ((double)(tend-tstart))/CLOCKS_PER_SEC;

    printf("El product escalar es %f \n\n", dot);
    printf("Tiempo de cpu (CLOCK) : %12.5f secs\n",cpu_time_used);
    printf("Tiempo (gettimeofday) = %12.5f secs\n",dt1/TUNIT);
    printf("Tiempo (MPI_WTime) = %12.5f secs\n",finish-start);

  MPI_Finalize();
}  /* main */
```

Teniendo como base este código ejemplo:

1) Compila el código anterior con mpicc y ejecútalo con mpirun usando un proceso y con 2 procesos.

2) Modifica el código para que se ejecute en paralelo. Utiliza la función colectiva MPI_Bcast para distribuir el tamaño de los vectores n, donde n será múltiplo del número de procesos p. Utiliza la función MPI_Scatter para distribuir los vectores x e y, que se encontrarán en el proceso "0", entre todos los procesos. Utiliza la función MPI_Reduce para recoger y sumar los resultados parciales obtenidos y calcular el resultado final del producto.
3) Ejecuta el código para tamaños: 80000, 800000, 8000000 y 80000000; y ejecútalo con 1, 2, 4, 8 y 12 procesos. Obtén una tabla de tiempos y una gráfica de aceleración para los diferentes tamaños. Ejecútalo en los siguientes casos:
    a. Todos los procesos se ejecutan en el host (boes.uv.es)
    b. Cada proceso se ejecuta en un nodo, comenzando por el host.
4) En los datos anteriores indica a partir de qué tamaños es interesante paralelizar el código y para qué número de procesadores.

### 3.5 Multiplicación matriz vector (Opcional: se entregará y presentará individualmente al profesor, aunque se puede realizar en grupo)

En este apartado vamos a tratar las funciones colectivas que sirven para recoger una estructura de datos entre los procesos y para ello utilizaremos la función MPI_Gather.

Se va a realizar la multiplicación paralela de una matriz (PxN) con un vector (Nx1), de números flotantes en doble precisión, donde P es el número de procesadores y N un valor fijado por el usuario. En esta formulación paralela del algoritmo, descomponemos la matriz A en filas, una por cada proceso. Por tanto, cada proceso deberá tener una fila de la matriz y el vector por el que se va a multiplicar. Los pasos a seguir son los siguientes:

1. El proceso raíz (0) genera la matriz A y el vector x.
2. La matriz A se distribuye entre los procesos (MPI_Scatter).
3. El vector se difunde entre los procesos (MPI_Bcast).
4. Cada proceso realiza la multiplicación escalar con los datos que tiene (su fila y el vector).
5. Se recoge el vector solución en el proceso raíz (0) mediante MPI_Gather.

Algunas consideraciones que debes tener en cuenta a la hora de realizar el código son las siguientes:

- El número de filas será igual al número de procesadores, por lo que la reserva de memoria de la matriz se realizará en tiempo de ejecución.
- Solo el proceso raíz (0) necesita tener toda la matriz A, mientras que el resto de procesos solo necesita tener una fila de la matriz.
- Cada proceso obtendrá como resultado un número, que al juntarlos con (MPI_Gather) en el proceso raíz (0) forman el vector solución.
- Podemos tomar tiempos mediante MPI_Wtime, pero hay que considerar que los tiempos son locales a un proceso. Por ello, interesa tomar tiempos asegurando que todos los procesos han tardado lo mismo.

Teniendo como base esta estructura ejemplo:

1) Diseña el código que resuelva esta aplicación mediante MPI
2) Ejecuta el código para 1, 2, 4, 6, 12, 16 procesos y comprueba para N= 1000,10000, 100000, 1000000 y 10000000 (o el límite de la máquina) las prestaciones en Mflops. Representa la gráfica resultante y comenta la escalabilidad de la aplicación en relación a la arquitectura del computador y al ejecutarlo sobre todas las máquinas disponibles.

**Nota:** al mantener constante la carga de trabajo por cada núcleo estás haciendo un estudio de escalabilidad

## 4. Evaluación de la práctica

La evaluación de esta sesión se realizará en la siguiente sesión mediante un examen constituido por preguntas de tipo test y de respuesta corta que tendrá una duración de 30 minutos.