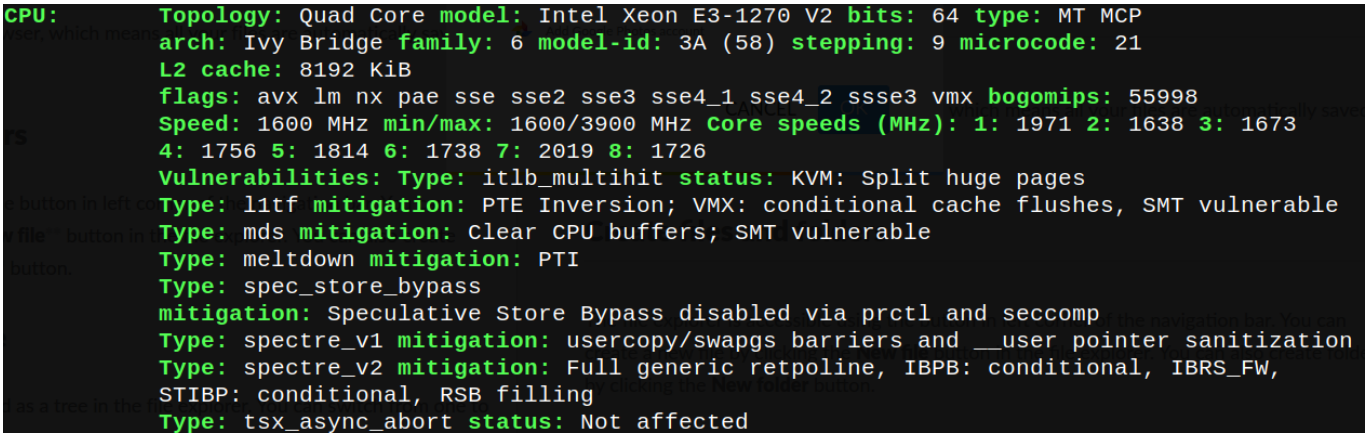
Computación en la Nube

Antes de realizar los ejercicios que están expuestos en la práctica se muestra el procesador sobre el cual se ejecutará los programas.



Ejercicio 1

Analiza el programa hello.c y realiza las siguientes ejecuciones comprobando en cada caso el resultado obtenido.

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

int  main(int argc,  char  **argv) {
 int rank, size;
 int namelen;
 char  processor_name[MPI_MAX_PROCESSOR_NAME];

 MPI_Init(  &argc,  &argv );
 MPI_Comm_size( MPI_COMM_WORLD,  &size );
 MPI_Comm_rank( MPI_COMM_WORLD,  &rank );
 MPI_Get_processor_name(processor_name,  &namelen);
 printf(  "Hello world from process %d of %d in %s\n", rank, size,
processor_name );
 MPI_Finalize();
}
```

Análisis

A continuación se realizará una breve descripción de las funciones que aparecen en `hello.c`:

- `MPI_Init`: Inicializa la estructura de comunicación de *MPI* entre los procesos.
- `MPI_Comm_size`: Determina el tamaño del comunicador seleccionado, es decir, el número de procesos que están actualmente asociados a este.
- `MPI_Comm_rank`: Determina el rango (identificador) del proceso que lo llama dentro del comunicador seleccionado.

- `MPI_Finalize`: Finaliza la comunicación paralela entre los procesos.

También se puede apreciar las siguientes constantes:

- `MPI_COMM_WORLD`: Identificador del comunicador al que pertenecen todos los procesos de una ejecución *MPI*
- `MPI_MAX_PROCESSOR_NAME`: Valor que determina la longitud máxima del nombre que puede tener un procesador.

Con todo esto en cuenta, podemos deducir que lo único que realiza el programa al ejecutarse es mostrar un mensaje `Hello World` por cada proceso que ha instanciado en la ejecución.

Resultado

Al ejecutamos un solo proceso del programa obtenemos la siguiente salida:

```
> mpirun -np 1 hello.run
Hello world from process 0 of 1 in ptondreau
```

Si ejecutamos varios procesos del programa, por ejemplo 4, obtenemos la siguiente salida:

```
> mpirun -np 4 hello.run
Hello world from process 0 of 4 in ptondreau
Hello world from process 1 of 4 in ptondreau
Hello world from process 2 of 4 in ptondreau
Hello world from process 3 of 4 in ptondreau
```

Ejercicio 2

Analiza y compila el programa helloms.

```
#include  <stdio.h>
#include  <string.h>
#include  "mpi/mpi.h"

int  main(int argc,  char  **argv)  {
 int rank, size, tag, rc, i;
 MPI_Status status;
 char  message[20];
 rc =  MPI_Init(&argc,  &argv);
 rc =  MPI_Comm_size(MPI_COMM_WORLD,  &size);
 rc =  MPI_Comm_rank(MPI_COMM_WORLD,  &rank);
 tag =  100;
 if(rank ==  0)  {
  strcpy(message,  "Hello, world");
  for  (i =  1; i < size; i++)
   rc =  MPI_Send(message,  13, MPI_CHAR, i, tag, MPI_COMM_WORLD);
 } else rc =  MPI_Recv(message,  13, MPI_CHAR,  0, tag, MPI_COMM_WORLD,
```

```
&status);

 printf(  "node %d : %.13s\n", rank, message);
 rc =  MPI_Finalize();
}
```

Análisis

A parte de las mismas funciones y constantes que se han visto en el programa `hello.c`, se hace uso de los siguientes elementos:

- `MPI_Status`: Elemento que almacena la información sobre operaciones de recepción demensajes.
- `MPI_Send`: Función que realiza el envío de un mensaje de un proceso fuente a otro destino.
- `MPI_Recv`: Rutina de recibimiento de un mensaje desde un proceso.

El programa `helloms.c` realiza lo siguiente:

1. Inicia las variables
2. Si el procesador es el procesador 0:
    - Almacena el mensaje `Hello, world` en la variables `message`.
    - Se envía el mensaje a cada procesador que esté disponible para el programa
3. En caso de no ser el procesador 0, espera para la recepción del mensaje
4. Muestra el mensaje en pantalla

Resultado

Ejecución del programa con 2 procesadores:

```
> mpirun -np 2 helloms.run
node 0 : Hello, world
node 1 : Hello, world
```

Ejecución del programa con 4 procesadores:

```
> mpirun -np 4 helloms.run
node 0 : Hello, world
node 2 : Hello, world
node 3 : Hello, world
node 1 : Hello, world
```

Ejercicio 3

Escribe un nuevo programa en el que los esclavos envían al maestro el mensaje y es el maestro el que muestra la salida.

Para realizar este ejercicio se usará el programa `helloms.c` como punto de partida y se modificará para cumplir lo que pide el ejercicio. El programa reescrito es el siguiente:

```
#include  <stdio.h>
#include  <string.h>
#include  "mpi/mpi.h"

int  main(int argc,  char  **argv)  {
 int rank, size, tag, rc, i;
 MPI_Status status;
 char  message[20];
 rc =  MPI_Init(&argc,  &argv);
 rc =  MPI_Comm_size(MPI_COMM_WORLD,  &size);
 rc =  MPI_Comm_rank(MPI_COMM_WORLD,  &rank);
 tag =  100;
 if(rank ==  0)  {
  for  (i =  1; i < size; i++)  {
   rc =  MPI_Recv(message,  13, MPI_CHAR, i, tag, MPI_COMM_WORLD,
&status);
   printf(  "node %d : %.13s\n", rank, message);
  }
 }
 else {
  strcpy(message,  "Hello, world");
  rc =  MPI_Send(message,  13, MPI_CHAR,  0, tag, MPI_COMM_WORLD);
 }

 rc =  MPI_Finalize();
}
```

Resultado

Ejecución del programa con 2 procesadores:

```
> mpirun -np 2 helloms.run
node 0 : Hello, world
```

- Solo se muestra un mensaje dado que existe un maestro y un esclado.

Ejecución del programa con 4 procesadores:

```
> mpirun -np 4 helloms.run
node 0 : Hello, world
node 0 : Hello, world
node 0 : Hello, world
```

- Se muestra tres mensajes dado que existen tres esclavos y un maestro.

Ejercicio 4

Escribe un programa que haga circular un token en un anillo

TODO

Ejercicio 5

El objetivo de este ejercicio es comprobar experimentalmente el costo de las comunicaciones entre pares de procesadores mediante ping-pong. Se trata además de comparar el coste de las comunicaciones con el coste de hacer una operación de tipo aritmético.

Programa `prod.c`

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

int  main(int argc,  char  *argv[]) {
 int myid, numprocs;
 double startwtime, endwtime, prodtime;
 int namelen;
 char  processor_name[MPI_MAX_PROCESSOR_NAME];
 double x, y, z;
 long  int i, iterations;

 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(stdout,"Process %d of %d on %s\n", myid, numprocs,
processor_name);

 y = z =  1;
 iterations =  1000000000;

 startwtime =  MPI_Wtime();
 for  (i =  0; i < iterations; i++)
  x = y * z + x;
 endwtime =  MPI_Wtime();

 prodtime =  (endwtime-startwtime)  /  (double) iterations;
 printf("wall clock time = %f, Prod time: %.16f, x = %f\n",  (endwtime-
startwtime), prodtime, x);

 MPI_Finalize();
 return  0;
}
```

La salida del programa `prod.c` nos mostrará el tiempo que se ha tardado para realizar un conjunto de operaciones, el tiempo que se ha tardado por operación y el número de operaciones que se han realizado.

En este caso, realiza 1000000000 operaciones aritméticas dadas por la función `x = y * z + x`

Resultado

La ejecución del programa se realiza con un único procesador y su salida es la siguiente:

```
> mpirun -np 1 prod.run
Process 0 of 1 on ptondreau
wall clock time = 2.354768, Prod time: 0.0000000023547679, x =
1000000000.000000
```

Programa `ptop.c`

```
#include  <stdio.h>
#include  <stdlib.h>
#include  "mpi/mpi.h"
#define  NUMBER_OF_TESTS  10

int  main(int argc,  char  **argv)  {
 double  *buf;
 int rank;
 int n;
 double t1, t2, tmin;
 int i, j, k, nloop;
 int my_name_length;
 char  my_name[BUFSIZ];
 MPI_Status status;

 MPI_Init(  &argc,  &argv );
 MPI_Comm_rank( MPI_COMM_WORLD,  &rank );
 MPI_Get_processor_name(my_name,  &my_name_length);
 printf("\nProcesador: %s", my_name);
 fflush(stdout);
 MPI_Barrier(MPI_COMM_WORLD);

 if  (rank ==  0) printf(  "\nKind\t\tn\ttime (sec)\tMB / sec\n"  );

 for  (n =  1; n <  1100000; n*=2)  {
  if  (n ==  0) nloop =  1000;
  else nloop =  1000  / n;

  if  (nloop <  1) nloop =  1;
  buf =  (double  *)  malloc( n *  sizeof(double)  );
  if  (!buf)  {
   fprintf( stderr, "Could not allocate send/recv buffer of size %d\n", n);
   MPI_Abort( MPI_COMM_WORLD,  1);
  }
  tmin =  1000;
  for  (k =  0; k < NUMBER_OF_TESTS; k++)  {
   if  (rank ==  0)  {
```

```
    /* Make sure both processes are ready */
    MPI_Barrier(MPI_COMM_WORLD);
    t1 =  MPI_Wtime();
    for  (j =  0; j < nloop; j++)  {
     MPI_Send( buf, n, MPI_DOUBLE,  1, k, MPI_COMM_WORLD );
     MPI_Recv( buf, n, MPI_DOUBLE,  1, k, MPI_COMM_WORLD, &status);
    }
    t2 =  (MPI_Wtime()  - t1)  / nloop;
    if  (t2 < tmin) tmin = t2;
   }
   else  if  (rank ==  1)  {
    /* Make sure both processes are ready */
    MPI_Barrier(MPI_COMM_WORLD);
    for  (j =  0; j < nloop; j++)  {
     MPI_Recv( buf, n, MPI_DOUBLE,  0, k, MPI_COMM_WORLD, &status);
     MPI_Send( buf, n, MPI_DOUBLE,  0, k, MPI_COMM_WORLD );
    }
   }
  }
  /* Convert to half the round-trip time */
  tmin = tmin /  2.0;
  if  (rank ==  0)  {
   double rate;
   if  (tmin >  0) rate = n *  sizeof(double)  *  1.0e-6  /tmin;
   else rate =  0.0;

   printf(  "Send/Recv\t%d\t%f\t%f\n", n, tmin, rate );
  }
  free( buf );
 }

 MPI_Finalize(  );
 return  0;
}
```

El programa `ptop.c` realizará la comunicación de paso de mensajes etre dos procesadores. Realizará operaciones de envío y recepción de datos, dichas operaciones se realizarán con un número variable de tamaño de datos que aumenta a cada iteración, obtendrá el tiempo mínimo de cada iteración y finalmente mostrará por pantalla lo siguiente:

- El tamaño del elemento
- El tiempo que ha tardado en realizarse la operación
- La velocidad de la transmisión de datos

Resultado

La ejecución del programa se realiza con dos procesadores y su salida es la siguiente:

```
> mpirun -np 2 ptop.run

Procesador: ptondreau 1
```

```
Procesador: ptondreau 0
Kind            n       time (sec)      MB / sec
Send/Recv       1       0.000001        9.581883
Send/Recv       2       0.000001        19.508626
Send/Recv       4       0.000001        39.152256
Send/Recv       8       0.000001        69.070612
Send/Recv       16      0.000001        133.631940
Send/Recv       32      0.000001        252.662411
Send/Recv       64      0.000001        420.108305
Send/Recv       128     0.000002        579.396187
Send/Recv       256     0.000002        921.208461
Send/Recv       512     0.000006        642.358651
Send/Recv       1024    0.000008        1067.083498
Send/Recv       2048    0.000010        1609.035095
Send/Recv       4096    0.000014        2403.403262
Send/Recv       8192    0.000021        3077.529919
Send/Recv       16384   0.000036        3632.564268
Send/Recv       32768   0.000065        4010.126898
Send/Recv       65536   0.000124        4239.942746
Send/Recv       131072  0.000238        4413.737370
Send/Recv       262144  0.000474        4425.239551
Send/Recv       524288  0.001021        4106.967287
Send/Recv       1048576 0.002206        3802.968122
```