Examen MPI

1.

```
/* 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;
}
```

1) Comenta en el código ejemplo qué hace cada una de las funciones de MPI invocadas.

MPI_INIT: inicia

MPI_Finalize();: Acaba

MPI_Comm_rank: Nos dice el número del proceso.

MPI_Comm_size: Nos dice cuantos procesos tenemos.

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

mpirun -np 12 ./p

```
int main( 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(hotsname,255);
If (rank%2==0)
        Printf("PAR
{
```

4) Añade el resto de máquinas del clúster al fichero maquinas.
5) ¿Qué ocurre si se ejecuta con la opción –npernode 1 y sin la opción -np? Y si la cambiamos por -npernode 2?
6) Si ponemos -npernode 1 y la opción -np, ¿funciona siempre? ¿Por qué?

4) creamos un fichero

Ejecutamos mpirun -np 6 -hostfile maquinetes ./p

Si no se pone nada slots es 1

boe.uv.es

Ejemplo del profesor:

boe.uv.es slots= 2

compute-0-0 slots = 2

compute-0-1 slots = 2

Salida

Hola desde el proceso 0 de 6 en el nodo boe.uv.es

Hola desde el proceso 1 de 6 en el nodo boe.uv.es

Hola desde el proceso 2 de 6 en el nodo compute-0-0

Hola desde el proceso 3 de 6 en el nodo compute-0-0

Hola desde el proceso 4 de 6 en el nodo compute-0-1

Hola desde el proceso 5 de 6 en el nodo compute-0-1

Ejecutamos mpirun -np 8 -hostfile maquinetes ./p

Da la vuelta vuelve al primer host

Mpirun -np 4 -npernode 2 -hostfile maquinetes ./p

Npernode sobreescribe los slots del fichero

El npernode es hard, si se acaban los slots da error

Mpirun -npernode 2 -hostfile maquinetes ./p

Ejecuta 6.

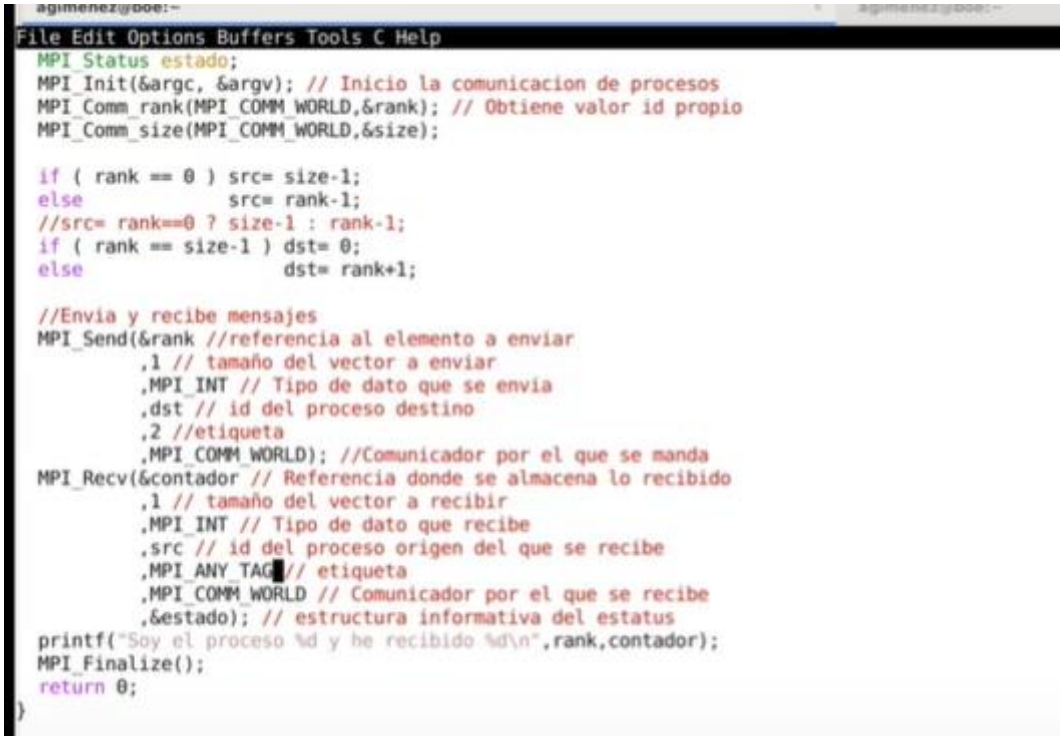
2)

```
#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;
  }
```

.

"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)

```
#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); fflush(stdout);
  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;
}
```

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.

```c
#include <mpi.h>
#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,i,rank,size,bsize,remain,beg,end;
  double x;

  MPI_Init ( &argc, &argv );
  MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
  MPI_Comm_size ( MPI_COMM_WORLD, &size );

  if ( rank == 0 )
    {
      printf("Introduce la precisión del cálculo (n > 0): ");
      fflush(stdout);
      scanf("%d",&n);
    }
  MPI_Bcast ( &n, 1, MPI_INT, 0, MPI_COMM_WORLD );
  bsize= n/size;
  remain= n%size;
  if ( rank == 0 )
    {
      beg= 0;
      end= bsize + remain;
    }
  else
```

4)

```c
/* 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 */
```

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.

```c
int main(int argc, char* argv[])
{
  double *x, *y,*x_local,*y_local;
  double dot, local_dot;
  int i, n,rank,size,n_local;
  double start, finish, dt1;
  struct timeval tv1, tv2;
  clock_t tstart,tend;
  double cpu_time_used;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
  MPI_Comm_size ( MPI_COMM_WORLD, &size );

  if ( rank == 0 )
    {

      if (argc<2){
        printf("Entra el número de elementos de cada vector: \n");
        fflush ( stdout );
        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));
```

```
    else
      n=atoi(argv[1]);
    // Reserva de espacio para los vectores
    x = (double *) malloc(n*sizeof(double));
    y = (double *) malloc(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();
  }
MPI_Bcast ( &n, 1, MPI_INT, 0, MPI_COMM_WORLD );
n_local= n/size;
if ( rank == 0 ) n_local+= n%size;
x_local= (double *) malloc ( sizeof(double)*n_local);
y_local= (double *) malloc ( sizeof(double)*n_local);
if ( rank == 0 )
  {
    MPI_Scatter ( &x[n%size], n/size, MPI_DOUBLE,
                  &x_local[n%size], n/size, MPI_DOUBLE,
                  0, MPI_COMM_WORLD );
    for ( i= 0; i < n%size; ++i )
      x_local[i]= x[i];
    MPI_Scatter ( &y[n%size], n/size, MPI_DOUBLE,
                  &y_local[n%size], n/size, MPI_DOUBLE,
                  0, MPI_COMM_WORLD );
    for ( i= 0; i < n%size; ++i )
-UUU:----F1  ej4.c          30% L60    (C/l Abbrev)----------------------
```

```
  }
MPI_Bcast ( &n, 1, MPI_INT, 0, MPI_COMM_WORLD );
n_local= n/size;
//if ( rank == 0 ) n_local+= n%size;
x_local= (double *) malloc ( sizeof(double)*n_local);
y_local= (double *) malloc ( sizeof(double)*n_local);
MPI_Scatter ( &(x[0]), n_local, MPI_DOUBLE,
              &(x_local[0]), n_local, MPI_DOUBLE,
              0, MPI_COMM_WORLD );
MPI_Scatter ( &(y[0]), n_local, MPI_DOUBLE,
              &(y_local[0]), n_local, MPI_DOUBLE,
              0, MPI_COMM_WORLD );
/*
if ( rank == 0 )
  {
    MPI_Scatter ( &x[n%size], n/size, MPI_DOUBLE,
                  &x_local[n%size], n/size, MPI_DOUBLE,
-UUU:**--F1  ej4.c          28% L57    (C/l Abbrev)----------------
```