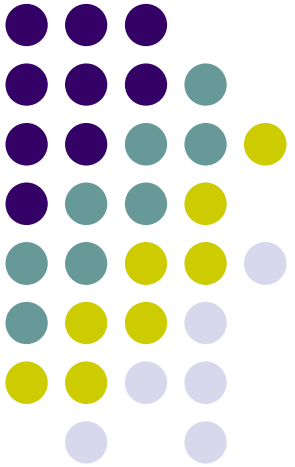
Curso de Promoción Educativa

Programación en el Supercomputador Ben Arabi

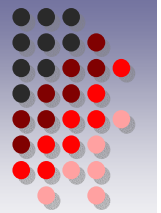
Programación con MPI

Javier Cuenca

Universidad de Murcia

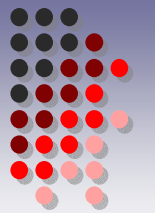


Indice de la sesión



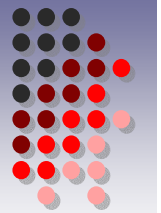
- Introducción
- Comunicaciones punto a punto
- Comunicaciones colectivas
- Agrupación de datos
- Comunicadores y topologías
- MPI-2
- Uso de MPI en el supercomputador BenArabi
- Ejercicios prácticos

Indice de la sesión



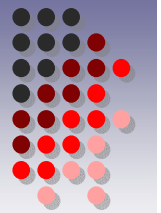
- Introducción.
  - Nociones básicas
  - Ejemplo inicial
  - Directivas
- Constructores
- Claúsulas de alcance de datos
- Funciones de librería
- Variables de entorno
- Ejemplos
- Tareas en OpenMP 3.0
- Uso de OpenMP en el supercomputador Ben Arabi
- Ejercicios prácticos

Introducción



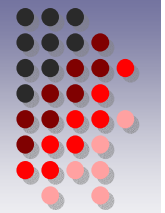
- Previamente PVM: Parallel Virtual Machine
- MPI: Message Passing Interface
- Una especificación para paso de mensajes
- La primera librería de paso de mensajes estándar y portable
- Por consenso MPI Forum. Participantes de unas 40 organizaciones
- Acabado y publicado en mayo 1994. Actualizado en junio 1995
- MPI-2, HeteroMPI, FT-MPI

Introducción.
¿Qué ofrece?



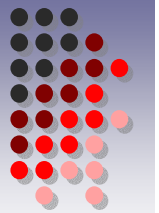
- Estandarización
- Portabilidad: multiprocesadores, multicomputadores, redes, heterogéneos, ...
- Buenas prestaciones, ..., si están disponibles para el sistema
- Amplia funcionalidad
- Implementaciones libres (mpich, lam, ...)

Introducción.
Procesos



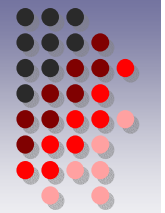
- Programa MPI: conjunto de procesos autónomos
- Cada proceso puede ejecutar codigo diferente
- Procesos comunican vía primitivas MPI
- Proceso: secuencial o multithreads
- MPI no proveé mecanismos para situar procesos en procesadores. Eso es misión de cada implementación en cada plataforma
- MPI 2.0:
  - Es posible la creación/borrado de procesos dinámicamente durante ejecución
  - Es posible el acceso a memoria remota
  - Es posible la entrada/salida paralela

Introducción.
Ejemplo: hello.c



```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
main(int argc, char*argv[]) {

int name, p, source, dest, tag = 0;
char message[100];
MPI_Status status;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&name);
MPI_Comm_size(MPI_COMM_WORLD,&p);
```

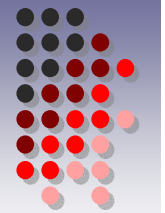
Introducción.
Ejemplo: hello.c



```
#include <stdio.h>
#in
#in
mai

int
cha
MPI

MPI
MPI
MPI
```

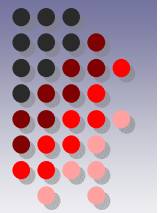
```
if (name != 0)
{
    printf("Processor %d of %d\n",name, p);
    sprintf(message,"greetings from process %d!", name);
    dest = 0;
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}
else
{
    printf("processor 0, p = %d \n",p);
    for(source=1; source < p; source++)
    {
      MPI_Recv(message,100, MPI_CHAR, source, tag,  MPI_COMM_WORLD, &status);
      printf("%s\n",message);
    }
  }
  MPI_Finalize();
}
```

Introducción.
Ejemplo FORTRAN: fortran_hello



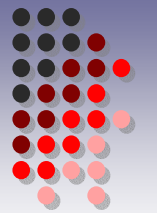
```
CHARACTER*20 msg
INTEGER myrank, ierr, status(MPI_STATUS_SIZE)
INTEGER tag = 99

…

CALL MPI_COMM_RANK( MPI_COMM_WORLD, myrank, ierr)

IF (myrank .EQ. 0) THEN
        msg = “Hello there”
        CALL MPI_SEND( msg, 11, MPI_CHARACTER, 1, tag, MPI_COMM_WORLD, ierr)

ELSE IF (myrank .EQ. 1) THEN
        CALL MPI_RECV( msg, 20, MPI_CHARACTER, 0, tag, MPI_COMM_WORLD, status,
ierr)

END IF
```

Introducción.
Ejemplo de uso



- Fichero cabecera:
  `#include <mpi.h>`
- Formato de las funciones:
  `error=MPI_nombre(parámetros ...)`
- Inicialización:
  `int  MPI_Init ( int  *argc , char  **argv )`
- Comunicador: Conjunto de procesos en que se hacen comunicaciones
  `MPI_COMM_WORD` , el mundo de los procesos MPI

Introducción.
Ejemplo de uso



- Identificación de procesos:

```
MPI_Comm_rank ( MPI_Comm comm , int  *rank)
```

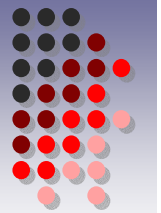
- Procesos en el comunicador:

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

- Finalización:

```
int  MPI_Finalize ( )
```

Introducción.
Ejemplo de uso

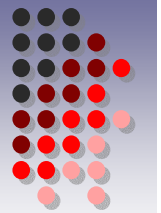


- MENSAJE: Formado por un cierto número de elementos de un tipo MPI
- Tipos MPI Básicos:

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_SHOT	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	
MPI_PACKED	

- Tipos MPI Derivados: los construye el programador

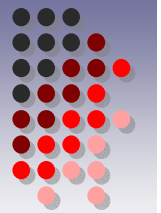
Introducción.
FORTRAN. Ejemplo de uso



- Tipos MPI Básicos
  - MPI_INTEGER — INTEGER
  - MPI_REAL — REAL
  - MPI_DOUBLE_PRECISION — DOUBLE PRECISION
  - MPI_COMPLEX — COMPLEX
  - MPI_LOGICAL — LOGICAL
  - MPI_CHARACTER — CHARACTER(1)
  - MPI_BYTE

- MPI_PACKED

Introducción.
Ejemplo de uso



- Envío:

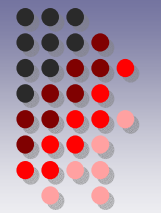
```
int MPI_Send ( void  *buffer , int contador , MPI_Datatype
   tipo , int destino , int tag , MPI_Comm  comunicador )
```

- Recepción:

```
int MPI_Recv ( void  *buffer , int contador , MPI_Datatype
   tipo , int origen , int tag , MPI_Comm  comunicador ,
   MPI_Status  *estado)
```

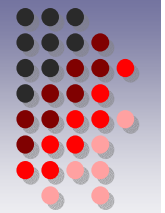
```
MPI_ANY_TAG
MPI_ANY_SOURCE
```

Indice de la sesión



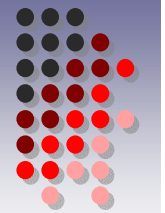
- Introducción
- **Comunicaciones punto a punto**
- Comunicaciones colectivas
- Agrupación de datos
- Comunicadores y topologías
- MPI-2
- Uso de MPI en el supercomputador BenArabi
- Ejercicios prácticos

Comunicaciones punto a punto.
Tipos de comunicaciones sincronas



- Envío:
  - Envío síncrono: `MPI_Ssend`
    Acaba cuando la recepción empieza
  - Envío con buffer: `MPI_Bsend`
    Acaba siempre, independiente del receptor
  - Envío estándar: `MPI_Send`
    Síncrono o con buffer
  - Envío "ready": `MPI_Rsend`
    Acaba independiente de que acabe la recepción
- Recepción: `MPI_Recv`
  Acaba cuando se ha recibido un mensaje.

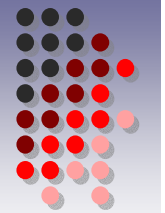
Comunicaciones punto a punto. Comunicación asíncrona (nonblocking)



- `MPI_Isend(buf, count, datatype, dest, tag, comm, request)`
- `MPI_Irecv(buf, count, datatype, source, tag, comm, request)`

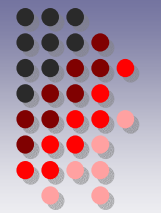
  Parámetro `request` para saber si la operación ha acabado
- `MPI_Wait( )`
  vuelve si la operación se ha completado. Espera hasta que se completa
- `MPI_Test( )`
  devuelve un flag diciendo si la operación se ha completado

Comunicaciones punto a punto. Comunicación asíncrona (nonblocking). Ejemplo: hello_nonblocking.c (1/2)



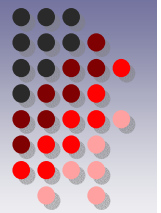
```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&name);
MPI_Comm_size(MPI_COMM_WORLD,&p);

 p_requests = (MPI_Request *) malloc ( p * sizeof(MPI_Request) );

if (name != 0)
{
   sprintf(message,"greetings from process %d!",name);
   dest = 0;
   MPI_Isend(message,    strlen(message)+1,    MPI_CHAR,    dest,    tag,
   MPI_COMM_WORLD, &request);
   printf("Procesador %d ya ha hecho el ISEND al procesador 0\n",name);


   /* …. Código por aquí enmedio …*/

   MPI_Wait(&request,&status);
   printf("Procesador %d ya ha pasado el WAIT tras envio\n",name);
}
```

Comunicaciones punto a punto. Comunicación asíncrona (nonblocking). Ejemplo: hello_nonblocking.c (2/2)



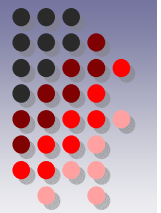
```
else
{
  for(source=1; source < p; source++)
  {
    MPI_Irecv(messages[source],100,MPI_CHAR,MPI_ANY_SOURCE,tag,
   MPI_COMM_WORLD, &p_requests[source]);
    printf("Proc.    0    ya    ha    hecho    IRECV    para    recibir    de
   source=%d\n\n",source);
  }

   /* …. Código por aquí enmedio …*/


  for(source=1; source < p; source++)
  {
    MPI_Wait(&p_requests[source],&status);
    printf("Tras el Wait del Receive: %s\n",messages[source]);
  }

}
free(p_requests);
MPI_Finalize();
}
```

Indice de la sesión



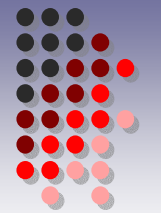
- Introducción
- Comunicaciones punto a punto
- **Comunicaciones colectivas**
- Agrupación de datos
- Comunicadores y topologías
- MPI-2
- Uso de MPI en el supercomputador BenArabi
- Ejercicios prácticos

Comunicaciones colectivas



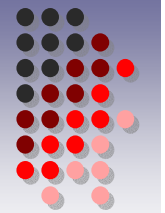
- `MPI_Barrier( )`
  bloquea los procesos hasta que la llaman todos
- `MPI_Bcast( )`
  broadcast del proceso raíz a todos los demás
- `MPI_Gather( )`
  recibe valores de un grupo de procesos
- `MPI_Scatter( )`
  distribuye un buffer en partes a un grupo de procesos
- `MPI_Alltoall( )`
  envía datos de todos los procesos a todos
- `MPI_Reduce( )`
  combina valores de todos los procesos
- `MPI_Reduce_scatter( )`
  combina valores de todos los procesos y distribuye
- `MPI_Scan( )`
  reducción prefija (0,...,i-1 a i)

Comunicaciones colectivas. broadcast.c



```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&p);

if(my_rank==0)
{
   printf("Introduce el dato:  a (float): ");scanf("%f",&a);
    pintf("Introduce    el    dato:         b    (float):    ");
        scanf("%f",&b);
   printf("Introduce    el    dato:         n    (entero):    ");
        scanf("%d",&n);
}

MPI_Bcast(&a,1,MPI_FLOAT,root,MPI_COMM_WORLD);
MPI_Bcast(&b,1,MPI_FLOAT,root,MPI_COMM_WORLD);
MPI_Bcast(&n,1,MPI_INT,root,MPI_COMM_WORLD);

if(my_rank !=0)
{
   printf("En procesador %d, los datos recibidos son a:%f    b:%f
   n:%d \n",my_rank,a,b,n);
}
MPI_Finalize();
```

Comunicaciones colectivas. broadcast.c



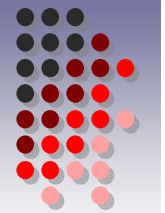
```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&p);
```

`MPI_Bcast(&a,1,MPI_FLOAT,root,MPI_COMM_WORLD);`

`&a:`	dirección de comienzo de buffer
`1:`	número de elementos en buffer
`MPI_FLOAT:`	tipo de datos del buffer
`root:`	identif. del root de la operación broadcast
`MPI_COMM_WORLD:`	comunicador

```
}
MPI_Finalize();
```

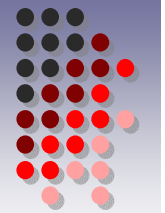
Comunicaciones colectivas. broadcast.c



```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&p);

if(my_rank==0)
{
   printf("Introduce el dato:  a (float): ");scanf("%f",&a);
    p
   pr
}
MPI_B
MPI_B
MPI_B

if(my_rank !=0)
{
   printf("En procesador %d, los datos recibidos son a:%f    b:%f
   n:%d \n",my_rank,a,b,n);
}
MPI_Finalize();
```

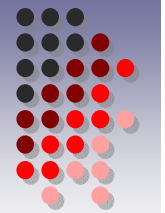
```
Introduce el dato:  a (float): 4.2
Introduce el dato:  b (float): 5.3
Introduce el dato:  n (entero): 6
En procesador 2, los datos recibidos son a:4.200000   b:5.300000   n:6
En procesador 1, los datos recibidos son a:4.200000   b:5.300000   n:6
En procesador 3, los datos recibidos son a:4.200000   b:5.300000   n:6
```

Comunicaciones colectivas.
gather.c



```
inicializa(my_rank,mis_datos,TAMA);
 if(my_rank==0)
        {
                datos=(int*)malloc(sizeof(int)*TAMA*p);
        }

  MPI_Gather(mis_datos,TAMA,MPI_INT,datos,TAMA,MPI_INT,root,MPI_C
  OMM_WORLD);
 if(my_rank==0)
        {
                printf("\n TODOS LOS DATOS RECIBIDOS EN PROCESO
  ROOT SON:\n");
                escribe(datos,TAMA*p);
                free(datos);
        }
```

Comunicaciones colectivas. gather.c



```
inicializa(my_rank,mis_datos,TAMA);
 if(my_rank==0)
```

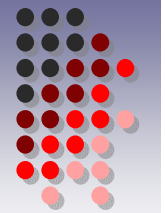
MPI_Gather(mis_datos,TAMA,MPI_INT,datos,TAMA,MPI_INT,root,MPI_COMM_WORLD);

Mis_datos:	dirección buffer de envío (en cada nodo emisor)
TAMA:	número de elementos a enviar desde cada proceso
MPI_INT:	tipo de datos de los elementos del buffer de envio
datos:	dirección buffer de recepción (en nodo receptor)
TAMA:	número de elementos de cada recepción individual
MPI_INT:	tipo de datos del buffer de recepción
root:	identificador del proceso que recibe los datos
MPI_COMM_WORLD:	comunicador

Com
gath

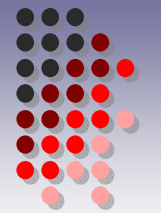
```
Datos iniciales en el proceso 1:

 10000    10001    10002    10003    10004    10005    10006    10007
10008    10009

 Datos iniciales en el proceso 2:

20000    20001    20002    20003    20004    20005    20006    20007    20008
20009

 Datos iniciales en el proceso 3:

30000    30001    30002    30003    30004    30005    30006    30007    30008
30009

 Datos iniciales en el proceso 0:

  0    1    2    3    4    5    6    7    8    9

 TODOS LOS DATOS RECIBIDOS EN PROCESO ROOT SON:

0    1    2    3    4    5    6    7    8    9    10000    10001    10002
10003    10004    10005    10006    10007    10008    10009    20000    20001
20002    20003    20004    20005    20006    20007    20008    20009    30000
30001    30002    30003    30004    30005    30006    30007    30008    30009
```

Comunicaciones colectivas. scatter.c



```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&p);

if(my_rank==0)
{
   datos=(int*)malloc(sizeof(int)*TAMA*p);
    inicializa(my_rank,datos,TAMA*p);
}

MPI_Scatter(datos,TAMA,MPI_INT,mis_datos,TAMA,MPI_INT,root,MPI_CO
   MM_WORLD)

printf("Datos recibidos por proceso %d son:\n", my_rank);
escribe(mis_datos,TAMA);


if(my_rank==0)
{
   free(datos);
}

MPI_Finalize();
```

Comunicaciones colectivas. scatter.c



```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&p);
```

MPI_Scatter(datos,TAMA,MPI_INT,mis_datos,TAMA,MPI_INT,root,MPI_COMM_WORLD)

datos: dirección buffer de envío

TAMA: número de elementos a enviar a cada proceso

MPI_INT: tipo de datos de los elementos del buffer de envio

mis_datos: dirección buffer de recepción

TAMA: número de elementos en buffer de recepción

MPI_INT: tipo de datos del buffer de recepción

root: identificador del proceso que envia datos

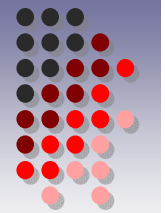
MPI_COMM_WORLD: comunicador

```
MPI_Finalize();
```

Comunicaciones colectivas. scatter.c

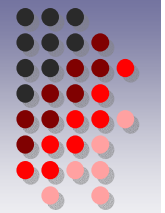
```
 Datos iniciales en proceso 0 son:

  0    1    2    3    4    5    6    7    8    9    10    11    12    13
14    15    16    17    18    19    20    21    22    23    24    25    26
27    28    29    30    31    32    33    34    35    36    37    38    39

Datos recibidos por proceso 1 son:

10    11    12    13    14    15    16    17    18    19

Datos recibidos por proceso 0 son:

  0    1    2    3    4    5    6    7    8    9

Datos recibidos por proceso 2 son:

 20    21    22    23    24    25    26    27    28    29

Datos recibidos por proceso 3 son:

30    31    32    33    34    35    36    37    38    39
```

Comunicaciones colectivas. reduce.c



```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&p);

   inicializa(my_rank,mis_datos,TAMA);

if(my_rank==0)
{
   datos=(int*)malloc(sizeof(int)*TAMA);
}


MPI_Reduce(mis_datos,datos,TAMA,MPI_INT,MPI_SUM,root,MPI_COMM_WORLD);

if(my_rank==0)
{
   printf("\n LOS DATOS, TRAS REDUCCION,  EN PROCESO ROOT SON:\n");
    escribe(datos,TAMA);
    free(datos);
}

MPI_Finalize();
```

Comunicaciones colectivas.
reduce.c

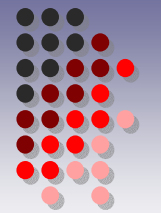


```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&p);
```

```
MPI_Reduce(mis_datos,datos,TAMA,MPI_INT,MPI_SUM,root,MPI_COMM_WORLD);

mis_datos:      dirección buffer de envío

datos:          dirección buffer de recepción

TAMA:           número de elementos en buffer de envío

MPI_INT:        tipo de elementos de buffer de envío

MPI_SUM:        operación a realizar durante la reducción

root:           identificador del proceso root

MPI_COMM_WORLD:comunicador
```

```
MPI_Finalize();
```

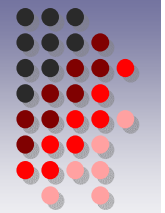
Comunicaciones colectivas.
reduce.c



```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_C

   in

if(my
{
   da
}

MPI_R

if(my
{
   pr
     e
     f
}

MPI_F
```

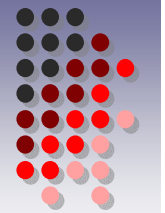
```
En proceso 2 los datos son:
  20000    20001    20002    20003    20004    20005    20006    20007
20008    20009


 En proceso 0 los datos son:
  0    1    2    3    4    5    6    7    8    9


 En proceso 1 los datos son:
  10000    10001    10002    10003    10004    10005    10006    10007
10008    10009


 LOS DATOS, TRAS REDUCCION,  EN PROCESO ROOT SON:
  30000    30003    30006    30009    30012    30015    30018    30021
30024    30027
```

Comunicaciones colectivas. Operaciones de reducción



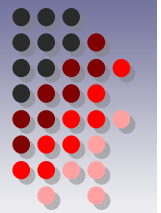
MPI Operator	Operation
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bitwise and
MPI_LOR	logical or
MPI_BOR	bitwise or
MPI_LXOR	logical exclusive or
MPI_BXOR	bitwise exclusive or
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

Comunicaciones colectivas.
FORTRAN



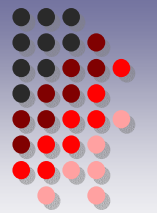
```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR

MPI_GATHER(SENDBUF,    SENDCOUNT,    SENDTYPE,    RECVBUF,
   RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER  SENDCOUNT,  SENDTYPE,  RECVCOUNT,  RECVTYPE,  ROOT,
   COMM, IERROR

MPI_SCATTER(SENDBUF,    SENDCOUNT,    SENDTYPE,    RECVBUF,
   RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER  SENDCOUNT,  SENDTYPE,  RECVCOUNT,  RECVTYPE,  ROOT,
   COMM, IERROR

MPI_REDUCE(SENDBUF,  RECVBUF,  COUNT,  DATATYPE,  OP,  ROOT,
   COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

Indice de la sesión



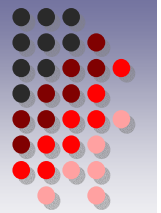
- Introducción
- Comunicaciones punto a punto
- Comunicaciones colectivas
- **Agrupación de datos**
- Comunicadores y topologías
- MPI-2
- Uso de MPI en el supercomputador BenArabi
- Ejercicios prácticos

Agrupación de datos



- Con contador en rutinas de envío y recepción: agrupar los datos a enviar
  - Con tipos derivados
  - Con empaquetamiento

Agrupación de datos. Tipos derivados

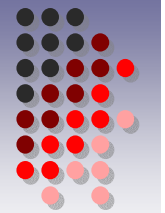


- Se crean en tiempo de ejecución
- Se especifica la disposición de los datos en el tipo:

```
Int MPI_Type_Struct
     (
   int                      count,
   int                      *array_of_block_lengths,
   MPI_Aint                 *array_of_displacements,
   MPI_Datatype             *array_of_types,
   MPI_Datatype             *newtype
   )
```

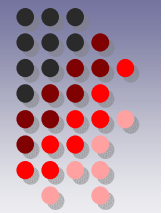
- Se pueden construir tipos de manera recursiva
- La creación de tipos requiere trabajo adicional

Agrupación de datos.
Tipos derivados: derivados.c (1/2)



```
typedef struct
{
        float   a;
        float   b;
        int     n;
} INDATA_TYPE;
INDATA_TYPE     indata;
MPI_Datatype    message_type;

if(my_rank==0)
{
   printf("Introduce el dato:a (float): ");scanf("%f",&(indata.a));
   printf("Introduce el dato:b (float): ");scanf("%f",&(indata.b));
   printf("Introduce el dato:n(entero): ");scanf("%d",&(indata.n));
}

Build_derived_type(&indata,&message_type);

MPI_Bcast(&indata,count,message_type,root,MPI_COMM_WORLD);

if(my_rank !=0)
   printf("En procesador %d, los datos recibidos son a:%f     b:%f
   n:%d \n",my_rank,indata.a,indata.b,indata.n)
```

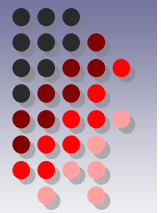
Agrupación de datos.
Tipos derivados: derivados.c (2/2)



```
void      Build_derived_type(INDATA_TYPE      *indata,      MPI_Datatype
   *message_type_ptr)
{
 int              block_lenghts[3];
 MPI_Aint         dis[3];
 MPI_Aint         addresses[4];
 MPI_Datatype     typelist[3];

 typelist[0]=MPI_FLOAT;
 typelist[1]=MPI_FLOAT;
 typelist[2]=MPI_INT;

 block_lenghts[0]=1;
 block_lenghts[1]=1;
 block_lenghts[2]=1;

 MPI_Address(indata,&addresses[0]);
 MPI_Address(&(indata->a),&addresses[1]);
 MPI_Address(&(indata->b),&addresses[2]);
 MPI_Address(&(indata->n),&addresses[3]);
       /*...continua...*/
```

```
/*...continuación...*/
dis[0]=addresses[1]-addresses[0];
dis[1]=addresses[2]-addresses[0];
dis[2]=addresses[3]-addresses[0];

MPI_Type_struct(3,block_lenghts,dis,
typelist,message_type_ptr);

MPI_Type_commit(message_type_ptr);
}
```

Agrupación de datos. Tipos derivados

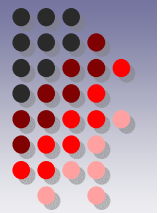


- Si los datos que constituyen el nuevo tipo son un subconjunto de entradas hay mecanismos especiales para construirlos (1/3):

```
int MPI_Type_contiguous
(
   int count,
   MPI_Datatype oldtype,
   MPI_Datatype *newtype
)
```

- Crea un tipo derivado formado por **`count`** elementos del tipo **`oldtype`** contiguos en memoria.

Agrupación de datos.
Tipos derivados

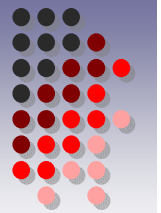


- Si los datos que constituyen el nuevo tipo son un subconjunto de entradas hay mecanismos especiales para construirlos (2/3):

```
int MPI_Type_vector
(
   int count,
   int block_lenght,
   int stride,
   MPI_Datatype element_type,
   MPI_Datatype *newtype
)
```

- Crea un tipo derivado formado por **`count`** elementos, cada uno de ellos con **`block_lenght`** elementos del tipo **`element_type`**.
- **`stride`** es el número de elementos del tipo **`element_type`** entre elementos sucesivos del tipo **`new_type`**.
- De este modo, los elementos pueden ser entradas igualmente espaciadas en un array.

Agrupación de datos. Tipos derivados

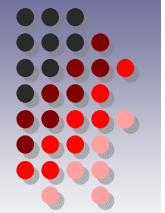


- Si los datos que constituyen el nuevo tipo son un subconjunto de entradas hay mecanismos especiales para construirlos (3/3):

```
int MPI_Type_indexed
(
   int count,
   int *array_of_block_lengths,
   int *array_of_displacements,
   MPI_Datatype element_type,
   MPI_Datatype *newtype
)
```

- Crea un tipo derivado con **`count`** elementos, habiendo en cada elemento **`array_of_block_lengths[i]`** entradas de tipo **`element_type`**, y el desplazamiento **`array_of_displacements[i]`** unidades de tipo **`element_type`** desde el comienzo de **`newtype`**

Agrupación de datos. Empaquetamiento



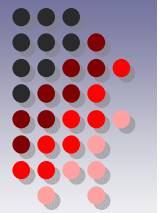
- Los datos se pueden empaquetar para ser enviados y desempaquetarse tras ser recibidos.

- Se empaquetan **in_count** datos de tipo **datatype**, y **pack_data** referencia los datos a empaquetar en el **buffer**, que debe consistir de **size** bytes (puede ser una cantidad mayor a la que se va a ocupar).

```
int MPI_Pack
(
void            *pack_data,
int             in_count,
MPI_Datatype    datatype,
void            *buffer,
int             size,
int             *position_ptr,
MPI_Comm        comm
)
```

- El parámetro **position_ptr** es de E/S:
  - Como entrada, el dato se copia en la posición **buffer+*position_ptr**.
  - Como salida, referencia la siguiente posición en el **buffer** después del dato empaquetado.
  - El cálculo de dónde se sitúa en el **buffer** el siguiente elemento a empaquetar lo hace MPI automáticamente.

Agrupación de datos. Empaquetamiento

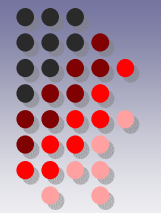


- Copia **count** elementos de tipo **datatype** en **unpack_data**, tomándolos de la posición **buffer+*position_ptr del buffer**.

- El tamaño del **buffer** (**size**) en bytes, y **position_ptr** es manejado por MPI de manera similar a como lo hace en MPI_Pack.

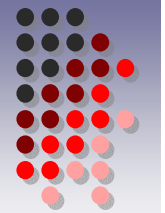
```
int MPI_Unpack
(
void              *buffer,
int               size,
int               *position_ptr,
Void              *unpack_data,
int               count,
MPI_Datatype      datatype,
MPI_Comm          comm
)
```

Agrupación de datos. Empaquetamiento: empaquetados.c



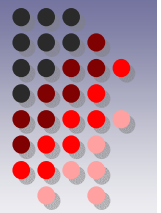
```
if(my_rank==0)
{
printf("Introduce     el     dato:          a     (float):     ");
   scanf("%f",&a);
printf("Introduce     el     dato:          b     (float):     ");
   scanf("%f",&b);
printf("Introduce     el     dato:          n     (entero):    ");
   scanf("%d",&n);
}
position=0;
MPI_Pack(&a,1,MPI_FLOAT,buffer,100,&position,MPI_COMM_WORLD);
MPI_Pack(&b,1,MPI_FLOAT,buffer,100,&position,MPI_COMM_WORLD);
MPI_Pack(&n,1,MPI_INT,buffer,100,&position,MPI_COMM_WORLD);

MPI_Bcast(buffer,position,MPI_PACKED,root,MPI_COMM_WORLD);

position=0;
MPI_Unpack(buffer,100,&position,&a,1,MPI_FLOAT,MPI_COMM_WORLD);
MPI_Unpack(buffer,100,&position,&b,1,MPI_FLOAT,MPI_COMM_WORLD);
MPI_Unpack(buffer,100,&position,&n,1,MPI_INT,MPI_COMM_WORLD);

printf("En procesador %d, los datos recibidos son a:%f    b:%f
   n:%d \n",my_rank,a,b,n);
```

Indice de la sesión



- Introducción
- Comunicaciones punto a punto
- Comunicaciones colectivas
- Agrupación de datos
- **Comunicadores y topologías**
- MPI-2
- Uso de MPI en el supercomputador BenArabi
- Ejercicios prácticos

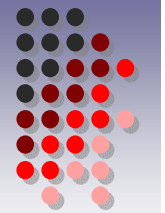
Comunicadores



- MPI_COMM_WORLD incluye a todos los procesos
- Se puede definir comunicadores con un número menor de procesos: para comunicar datos en cada fila de procesos en la malla, en cada columna, ...
- Dos tipos de comunicadores:

1. **intra-comunicadores:** se utilizan para enviar mensajes entre los procesos en ese comunicador,
2. **inter-comunicadores:** se utilizan para enviar mensajes entre procesos en distintos comunicadores.

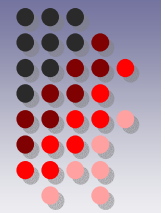
Comunicadores.
Intra-comunicadores



1. **Intra-comunicador**. Consta de:

   - **Un grupo**, que es una colección ordenada de procesos a los que se asocia identificadores entre 0 y *p*-1.
   - **Un contexto**, que es un identificador que asocia el sistema al grupo.
   - Adicionalmente, a un comunicador se le puede asociar **una topología virtual**.

Comunicadores.
Intra-comunicadores

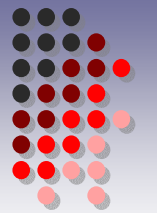


- MPI_COMM_WORLD consta de $p=q^2$ **procesos** agrupados en ***q*** **filas** y ***q*** **columnas**. El proceso número *r* tiene las coordenadas (*r* div *q*,*r* mod *q*).

- Ejemplo: Creación un comunicador cuyos procesos son los de la primera fila de nuestra malla virtual.

```
MPI_Group  MPI_GROUP_WORLD;
MPI_Group  first_row_group, first_row_comm;
int  row_size;
int  *process_ranks;

process_ranks=(int *) malloc(q*sizeof(int));

for(proc=0;proc<q;proc++)
   process_ranks[proc]=proc;

MPI_Comm_group(MPI_COMM_WORLD,&MPI_GROUP_WORLD);
MPI_Group_incl(MPI_GROUP_WORLD,q,process_ranks,&first_row_group);
MPI_Comm_create(MPI_COMM_WORLD,first_row_group,&first_row_comm);
```

Comunicadores.
Intra-comunicadores

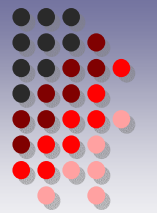


- Para crear varios comunicadores disjuntos

```
int MPI_Comm_split
(
        MPI_Comm old_comm,
        int split_key,
        int rank_key,
        MPI_Comm *new_comm
)
```

- Crea un nuevo comunicador para cada valor de **`split_key`**.
- Los procesos con el mismo valor de **`split_key`** forman un grupo.
- Si dos procesos ***a*** y ***b*** tienen el mismo valor de **`split_key`** y el **`rank_key`** de ***a*** es menor que el de ***b***, en el nuevo grupo ***a*** tiene identificador menor que ***b***.
- Si los dos procesos tienen el mismo **`rank_key`** el sistema asigna los identificadores arbitrariamente.

Comunicadores.
Intra-comunicadores

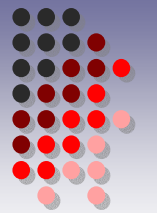


```
int MPI_Comm_split
        ( ...
        )
```

- Es una operación colectiva.
- Todos los procesos en el comunicador deben llamarla.
- Los procesos que no se quiere incluir en ningún nuevo comunicador pueden utilizar el valor `MPI_UNDEFINDED` en `rank_key`, con lo que el valor de retorno de `new_comm` es `MPI_COMM_NULL`.
- Ejemplo: crear *q* grupos de procesos asociados a las *q* filas:

```
MPI_Comm my_row_comm;
int my_row;

my_row=my_rank/q;
MPI_Comm_split (MPI_COMM_WORLD,my_row,my_rank,&my_row_comm);
```

Comunicadores.
Intra-comunicadores. Topologías



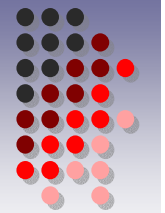
- A un grupo se le puede asociar una topología virtual:
  - topología de grafo en general
  - de malla o cartesiana

- Una topología cartesiana se crea:

```
int MPI_Card_create (
   MPI_Comm  old_comm,
   int  number_of_dims,
   int *dim_sizes,
   int *periods,
     int reorder,
     MPI_Comm *cart_comm    )
```

- El número de dimensiones de la malla es **`number_of_dims`**
- El número de procesos en cada dimensión está en **`dim_sizes`**
- Con **`periods`** se indica si cada dimensión es circular o lineal
- Un valor de 1 en **`reorder`** indica al sistema que se reordenen los procesos para optimizar la relación entre el sistema físico y el lógico.

Comunicadores.
Intra-comunicadores. Topologías



- Las coordenadas de un proceso conocido su identificador se obtienen con
```
int MPI_Cart_coords( MPI_Comm comm, int rank,
int number_of_dims, int *coordinates)
```

- El identificador, conocidas las coordenadas con
```
int MPI_Cart_rank( MPI_Comm comm, int *coordinates, int *rank)
```

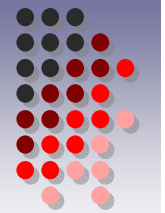
- Una malla se puede particionar en mallas de menor dimensión
```
int MPI_Cart_sub(MPI_Comm old_comm, int *varying_coords,
MPI_Comm *new_comm)
```

  - en **`varying_coords`** se indica para cada dimensión si pertenece al nuevo comunicador.

  - Si **`varying_coords`**`[0]=0` y **`varying_coords`**`[1]=1` para obtener el nuevo comunicador no se varía la primera dimensión pero sí la segunda. Se crean ***q*** comunicadores, uno por cada fila.

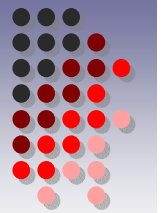
  - Es colectiva.

Indice de la sesión



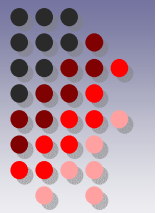
- Introducción
- Comunicaciones punto a punto
- Comunicaciones colectivas
- Agrupación de datos
- Comunicadores y topologías
- **MPI-2**
- Uso de MPI en el supercomputador BenArabi
- Ejercicios prácticos

MPI-2



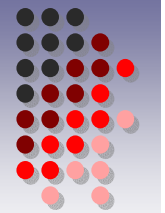
- **Corregir errores de MPI-1**
- **Entrada/Salida paralela (MPI-IO)**
  - Aumentar prestaciones E/S
  - Accesos no contiguos a memoria y a ficheros
  - Operaciones colectivas de E/S
  - Punteros a ficheros tantos individuales como colectivos
  - E/S asíncrona
  - Representaciones de datos portables y ajustadas a las necesidades
- **Operaciones remotas de memoria**
  - Proveer elementos del “tipo” de memoria compartida
  - Concepto de “ventana de memoria”: porción de memoria de un proceso que es expuesta explícitamente a accesos de otros procesos
  - Operación remotas son asíncronas → necesidad de sincronizaciones explícitas
- **Gestión dinámica de procesos**
  - Un proceso MPI puede participar en la creación de nuevos procesos: ***spawing***
  - Un proceso MPI puede comunicarse con procesos creados separadamente: ***connecting***
  - La clave: **intercomunicadores**: comunicadores que contienen 2 grupos de procesos
  - Extensión de comunicaciones colectivas a intercomunicadores

MPI-2
Entrada/Salida paralela (MPI-IO): escribiendo en diferentes ficheros: mpi_io_1.c



```
int main(int argc, char *argv[])
{
  int i, myrank, buf[BUFSIZE];
  char filename[128];
  MPI_File myfile;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

  for (i=0;i<BUFSIZE;i++)
    buf[i]=myrank*BUFSIZE+i;

  sprintf(filename,"testfile.%d",myrank);//cada proceso → un fichero

  MPI_File_open(MPI_COMM_SELF,filename, MPI_MODE_WRONLY | MPI_MODE_CREATE,
     MPI_INFO_NULL,&myfile);
  MPI_File_write(myfile,buf,BUFSIZE,MPI_INT,MPI_STATUS_IGNORE);
  MPI_File_close(&myfile);

  MPI_Finalize();
  return 0;
}
```

MPI-2
Entrada/Salida paralela (MPI-IO): escribiendo en diferentes ficheros: mpi_io_1.c



```
MPI_File_open(MPI_COMM_SELF,filename, MPI_MODE_WRONLY |
     MPI_MODE_CREATE, MPI_INFO_NULL,&myfile);
```

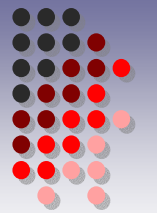
- MPI_COMM_SELF: Comunicador de los procesos que abren el fichero. En este caso cada proceso abre el suyo propio.
- Modo apertura
- Campo nulo
- Descriptor del fichero

```
MPI_File_write(myfile,buf,BUFSIZE,MPI_INT,MPI_STATUS_IGNORE);
```

- Descriptor del fichero
- Datos a escribir
- Tamaño
- Tipo
- MPI_STATUS_IGNORE: para que no me devuelva información en Status, porque no la necesito

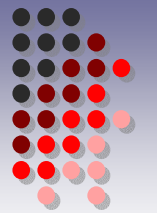
```
}
```

MPI-2
Entrada/Salida paralela (MPI-IO): escribiendo en un fichero único: mpi_io_2.c



```
int main(int argc, char *argv[])
{
  int i, myrank, buf[BUFSIZE];
  char filename[128];
  MPI_File myfile;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  for (i=0;i<BUFSIZE;i++)
   buf[i]=myrank*BUFSIZE+i;

  sprintf(filename,"testfile");//todos procesos → un único fichero
  MPI_File_open(MPI_COMM_WORLD,filename, MPI_MODE_WRONLY | MPI_MODE_CREATE,
      MPI_INFO_NULL,&myfile);
MPI_File_set_view(myfile,myrank*BUFSIZE*sizeof(int),MPI_INT,MPI_INT,”native”,MP
      I_INFO_NULL);
  MPI_File_write(myfile,buf,BUFSIZE,MPI_INT,MPI_STATUS_IGNORE);
  MPI_File_close(&myfile);

  MPI_Finalize();
  return 0;
}
```

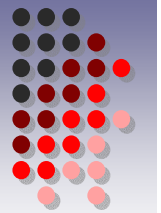
MPI-2
Entrada/Salida paralela (MPI-IO): escribiendo en un fichero único: mpi_io_2.c



```
MPI_File_set_view(myfile,myrank*BUFSIZE*sizeof(int),MPI_INT,MPI_INT,"
    native",MPI_INFO_NULL);
```

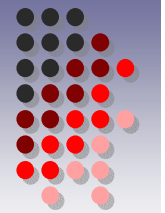
Descriptor del Fichero
Desplazamiento donde escribir
Tipo a escribir
Tipo zona discontinua del fichero
"native" representación del dato a escribir tal como se representa en memoria

MPI-2
Entrada/Salida paralela (MPI-IO):
leyendo de un fichero único: mpi_io_3.c



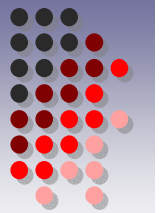
```
int main(int argc, char *argv[])
{
     /* … */

  MPI_File_open(MPI_COMM_WORLD,"testfile", MPI_MODE_RDONLY,
     MPI_INFO_NULL,&myfile);

  MPI_File_get_size(myfile,&filesize); //in bytes
  filesize=filesize/sizeof(int); //in number of ints
  bufsize=filesize/numprocs +1; //local number to read
  buf=(int *) malloc (bufsize*sizeof(int));

     MPI_File_set_view(myfile,myrank*bufsize*sizeof(int),MPI_INT,MPI_INT,"nativ
     e",MPI_INFO_NULL);
  MPI_File_read(myfile,buf,bufsize,MPI_INT,&status);
  MPI_Get_count(&status,MPI_INT,&count);

  printf("process %d read %d ints \n",myrank,count);

  /* … */

}
```

Indice de la sesión



- Introducción
- Comunicaciones punto a punto
- Comunicaciones colectivas
- Agrupación de datos
- Comunicadores y topologías
- MPI-2
- **Uso de MPI en el supercomputador BenArabi**
- Ejercicios prácticos

Uso de MPI en el supercomputador Ben Arabí



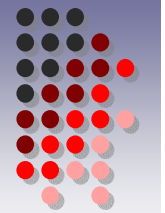
Ben Arabí cuenta con dos distribuciones de MPI

- Platform MPI (el propio de HP)
- Intel MPI

Antes de compilar y/o ejecutar: cargar las variables de entorno

- Para usar MPI de de platform:
  - `module unload impi`
  - `module load pmpi`
- Para usar MPI de de Intel:
  - `module unload pmpi`
  - `module load impi`

Uso de MPI en el supercomputador Ben Arabí
Compilación y ejecución con Intel MPI



Previo:

- `module unload pmpi`
- `module load impi`

Para compilar:

- `mpicc`: compila código MPI en C, utilizando el compilador GNU gcc
- `mpixx`: compila código MPI en C++, utilizando el compilador GNU gcc
- `mpif90`: compila código MPI en fortran 90, utilizando el compilador GNU gfortran
- `mpiicc`: compila código MPI en C, utilizando el compilador de Intel icc
- `mpiicpc`: compila código MPI en C++, utilizando el compilador de Intel icpc
- `mpifort`: compila código MPI en fortran 90, utilizando el compilador de intel ifort

Para enviar a la cola:

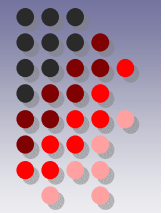
- `bsub –q nombre_cola –e fichero_salida_error –o fichero_salida_estandar –a intelmpi –n numero_cores ./nombre_ejecutable`

Ejemplo:

- `mpiicc ejemplo.c –o ejemplo_i`

- `bsub –q arabi_short –e ./%J.err –o ./%J.out –a intelmpi –n 8 mpirun.lsf ./ejemplo_i`

Uso de MPI en el supercomputador Ben Arabí
Compilación y ejecución con Intel MPI



Consejo: usar scripts para mandar trabajos paralelos a una cola.

Ejemplo:

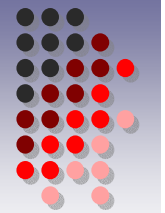
Se escribe el script `trabajo_ejemplo_i`:

```
#!/bin/bash
#BSUB -a intelmpi           # Se especifica el tipo de mpi
#BSUB -J ejemplo_i          # Nombre del trabajo
#BSUB -o ejemplo_i.%J.out   # Nombre del fichero de salida
#BSUB -e ejemplo_i.%J.err   # Nombre del fichero de error
#BSUB -q arabi_short        # Nombre de la cola
#BSUB -n 8                  # Número de cores a reservar

# Cargamos Entorno Necesario
source /etc/profile.d/modules.sh
module unload pmpi
module load impi

# Ejecución del trabajo
mpirun.lsf /users/$USER/ejemplo_i
```

Una vez creado el script del trabajo a ejecutar (`trabajo_ejemplo_i`) y con el programa ejemplo a ejecutar en dicho trabajo ya compilado (`ejemplo_i`), se manda el trabajo al sistema de colas:

```
bsub < trabajo_ejemplo_i
```

Uso de MPI en el supercomputador Ben Arabí
Compilación y ejecución con Platform MPI



Previo:

- `module unload impi`
- `module load pmpi`

Para compilar:

- `mpicc`: compila código MPI en C, utilizando el compilador de la plataforma
- `mpiCC`: compila código MPI en C++, utilizando el compilador de la plataforma
- `mpif77`: compila código MPI en fortran 77, utilizando el compilador de la plataforma
- `mpif90`: compila código MPI en fortran 90, utilizando el compilador de la plataforma

Para enviar a la cola:

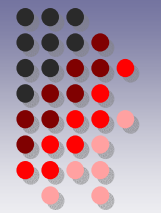
- `bsub –q nombre_cola –e fichero_salida_error –o fichero_salida_estandar –n numero_cores ./nombre_ejecutable`

Ejemplo:

- `mpicc ejemplo.c –o ejemplo_p`

- `bsub –q arabi_short –e ./%J.err –o ./%J.out –n 8 mpirun –lsb_hosts ./ejemplo_p`

Uso de MPI en el supercomputador Ben Arabí
Compilación y ejecución con Platform MPI



Consejo: usar scripts para mandar trabajos paralelos a una cola.

Ejemplo:

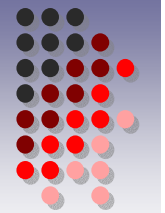
Se escribe el script `trabajo_ejemplo_p:`

```
#!/bin/bash

#BSUB -J ejemplo_p  # Nombre del trabajo
#BSUB -o ejemplo_p.%J.out      # Nombre del fichero de salida
#BSUB -e ejemplo_p.%J.err      # Nombre del fichero de error
#BSUB -q arabi_short           # Nombre de la cola
#BSUB -n 8                     # Número de cores a reservar

# Cargamos Entorno Necesario
source /etc/profile.d/modules.sh
module unload impi
module load pmpi

# Ejecución del trabajo
mpirun –lsb_hosts /users/$USER/ejemplo_p
```

Una vez creado el script del trabajo a ejecutar (`trabajo_ejemplo_p`) y con el programa ejemplo a ejecutar en dicho trabajo ya compilado (`ejemplo_p`), se manda el trabajo al sistema de colas:

```
bsub < trabajo_ejemplo_p
```

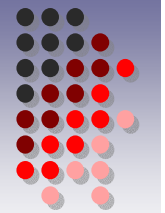
Uso de MPI en el supercomputador Ben Arabí
Compilación y ejecución.



Opciones más comunes del comando BSUB	
-J nombre_trabajo	Asigna un nombre al trabajo
-u email	Indica la dirección de correo
-B	Envía un correo al empezar el trabajo
-N	Envía un correo al finalizar el trabajo
-e fichero_error	Redirige stderr al fichero especificado
-o fichero_salida	Redirige stdout al fichero especificado
-W runtime	Fija el límite de tiempo wallclock
-q nombre_cola	Especifica la cola que se va a usar
-n num_core	Especifica el número de cores
-R	Especifica requerimientos de recursos

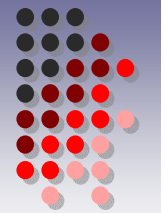
TABLA 5: OPCIONES DEL COMANDO BSUB

Uso de MPI en el supercomputador Ben Arabí
Compilación y ejecución



Nombre de la cola	Límite temporal	Número de cores
arabi_256x48h	48 horas	193-256
arabi_192x72h	72 horas	129-192
arabi_128x96h	96 horas	64-128
arabi_64x120h	120 horas	33-64
arabi_32x144h	144 horas	9-32
arabi_8x168h	168 horas	8
arabi_8x24h	24 horas	8
arabi_short	1 hora	Hasta 64

Uso de MPI en el supercomputador Ben Arabí
Compilación y ejecución.



- Ver las colas disponibles ejecutamos **`bqueues`**.

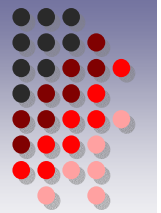
- Consultar el estado de nuestros trabajos ejecutamos el comando **`bjobs`**.
  - `PEND`: esperando en la cola para ser atendido.
  - `RUN`: enviado a un host y ejecutándose.
  - `USUSP`: suspendido por el usuario.
  - `PSUSP`: suspendido mientras esperaba ser atendido
  - `SSUSP`: suspendido por el sistema LSF

- Obtener información detallada: **`bjobs`** con la opción **`-l`**.
  - los recursos usados, los parámetros enviados, el entorno de ejecución, etc.

- Normalmente la salida de un trabajo no está disponible hasta que finaliza
  - Ver la salida hasta el momento con el comando **`bpeek`** seguido del **`job_id`**.
  - Si se ejecuta sin el **`job_id`** : información sobre el último trabajo lanzado.

- Eliminamos un trabajo con el comando **`bkill`** seguido del j**`ob_id`**

Uso de MPI en el supercomputador Ben Arabí
Compilación y ejecución.



Algunas ejemplos de opciones extras para ejecución:

Cambiar el número de procesos, respecto al número de cores:

Añadir "`-np numero_de_procesos`" a la línea del ejecución `mpirun` (solamente con Platform, no funciona con Intel).

Por ejemplo, 16 procesos:

```
mpirun –np 16 –lsb_hosts /users/$USER/ejemplo_p
```

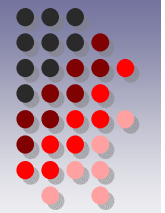
Forzar que el conjunto de todos los procesos se ejecuten en un solo nodo

```
#BSUB –R "span[hosts=1]"
```

Especificar que queremos *n* procesos por cada nodo

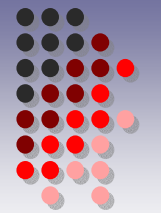
```
#BSUB –R "span[ptile=n]"
```

Uso de MPI en el supercomputador Ben Arabí
Comparación prestaciones con distintos entornos
Ejemplo: prestaciones_broadcast.c



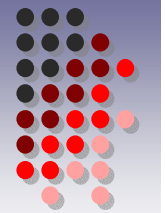
```
 …
MPI_Barrier(MPI_COMM_WORLD);
ti=MPI_Wtime();

MPI_Bcast(&n,tama,MPI_INT,root,MPI_COMM_WORLD);

MPI_Barrier(MPI_COMM_WORLD);
tf=MPI_Wtime();


if (my_rank==0)
{
  sprintf(fic,"res_hp_%d",p);
  fd=fopen(fic,"at");

  fprintf(fd,“Core %d. tama=%d. \t Tiempo=%lfsg\n",my_rank,tama,tf-ti);
  printf(“Core %d. tama=%d. \t Tiempo=%lf sg\n",my_rank,tama,tf-ti);

  fclose(fd);
}

…
```

Uso de MPI en el supercomputador Ben Arabí
Comparación prestaciones con distintos entornos
Ejemplo: trabajo_prestaciones_broadcast_hp



```
#!/bin/bash
#BSUB -J prestaciones_broadcast_hp                  # Nombre del trabajo
#BSUB -o prestaciones_broadcast_hp.%J.out           # Nombre del fichero de salida
#BSUB -e prestaciones_broadcast_hp.%J.err           # Nombre del fichero de error
#BSUB -q arabi_short                              # Nombre de la cola
#BSUB -n 4                                        # Nero de cores a reservar

# Cargamos Entorno Necesario
source /etc/profile.d/modules.sh
module unload impi
module load pmpi

# Ejecuci del trabajo
for tamanyo in 100000 200000 400000
do
          for i in 1 2 3 4 5
          do
                    mpirun  -np 4 -lsb_hosts
                    /users/$USER/ejemplos_mpi/dir_broadcast/
                    prestaciones_broadcast_hp $tamanyo
        done
done
```

Uso de MPI en el supercomputador Ben Arabí
Comparación prestaciones con distintos entornos
Ejemplo: trabajo_prestaciones_broadcast_hp



```
#!/bin/bash
#BSUB -J prestaciones_broadcast_hp                    # Nombre del trabajo
#BSUB -o presta                                       de salida
#BSUB -e presta                                       de error
#BSUB -q arabi_
#BSUB -n 4                                            r

# Cargamos Ento
source /etc/pro
module unload i
module load pmp

# Ejecuci del t
for tamanyo in
do
        for i i
        do



        done
done
```

```
> more res_hp_4
En procesador 0. tama=100000.    Tiempo=0.003430 sg
En procesador 0. tama=100000.    Tiempo=0.001419 sg
En procesador 0. tama=100000.    Tiempo=0.003753 sg
En procesador 0. tama=100000.    Tiempo=0.003879 sg
En procesador 0. tama=100000.    Tiempo=0.001291 sg

En procesador 0. tama=200000.    Tiempo=0.002619 sg
En procesador 0. tama=200000.    Tiempo=0.005780 sg
En procesador 0. tama=200000.    Tiempo=0.004301 sg
En procesador 0. tama=200000.    Tiempo=0.004419 sg
En procesador 0. tama=200000.    Tiempo=0.002629 sg

En procesador 0. tama=400000.    Tiempo=0.004525 sg
En procesador 0. tama=400000.    Tiempo=0.004564 sg
En procesador 0. tama=400000.    Tiempo=0.005627 sg
En procesador 0. tama=400000.    Tiempo=0.005233 sg
En procesador 0. tama=400000.    Tiempo=0.005031 sg
```

Uso de MPI en el supercomputador Ben Arabí
Comparación prestaciones con distintos entornos
Comunicaciones punto a punto

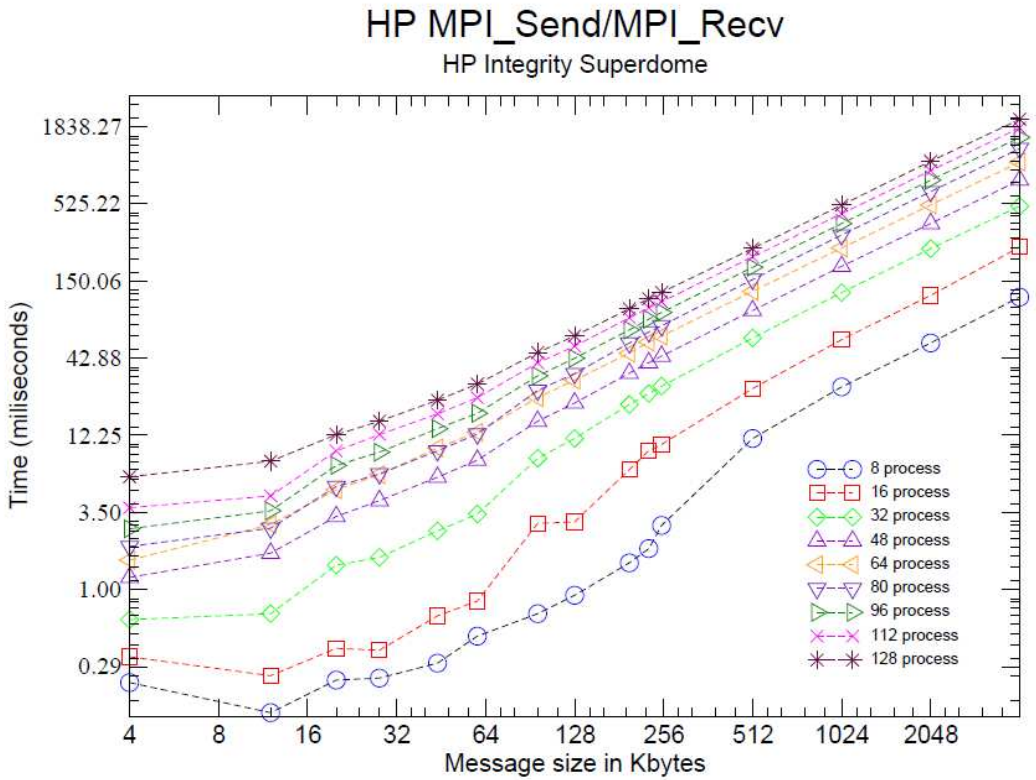
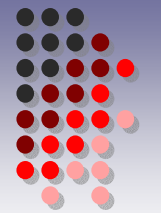


Figura 3: MPI_Send-MPI_Recv con HP-MPI para diferentes tamaños de mensaje en el HP Integrity Superdome.

Uso de MPI en el supercomputador Ben Arabí
Comparación prestaciones con distintos entornos
Comunicaciones punto a punto



MPI_Send-MPI_Recv con HP-MPI en el Superdome Ben

- *n* >12 KB se modifica pendiente curva ← cambio protocolo
- Tiempos con 64 cores = tiempos con 80 cores para varios *n*

Uso de MPI en el supercomputador Ben Arabí
Comparación prestaciones con distintos entornos
Comunicaciones punto a punto

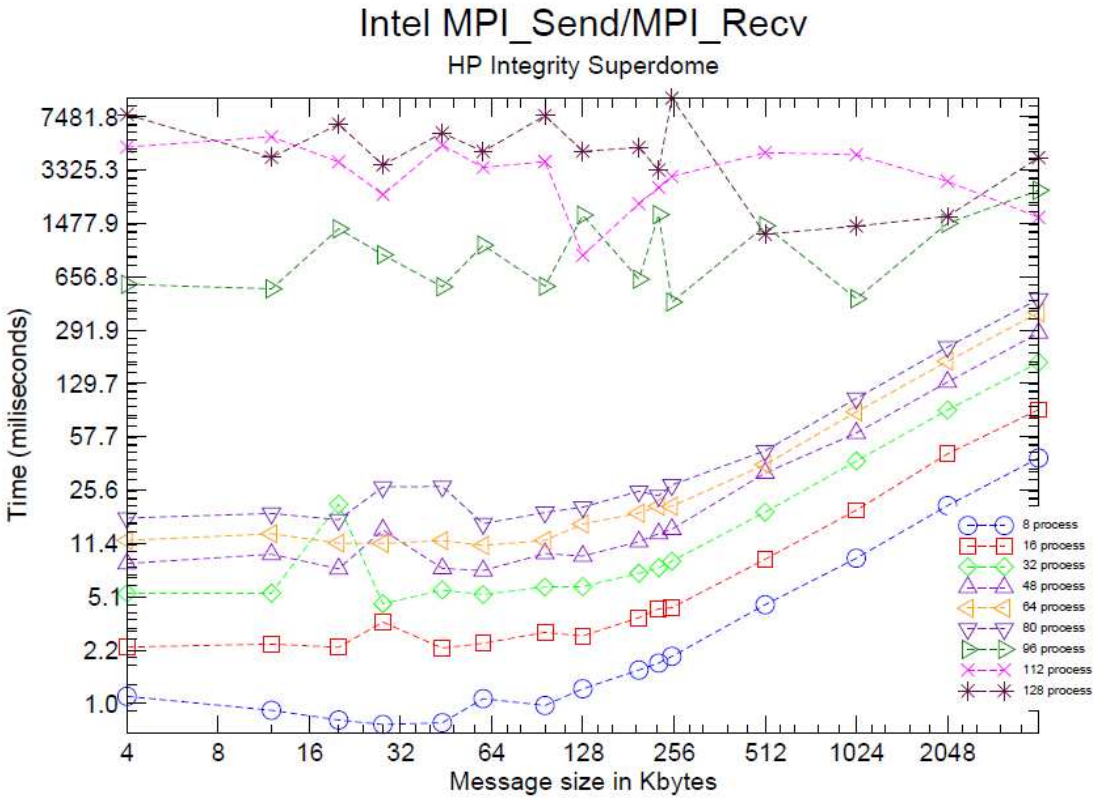
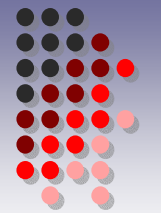


Figura 4: MPI_Send-MPI_Recv con Intel MPI para diferentes tamaños de mensaje en el HP Integrity Superdome.

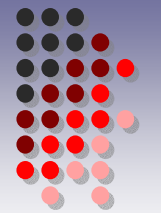
Uso de MPI en el supercomputador Ben Arabí
Comparación prestaciones con distintos entornos
Comunicaciones punto a punto



MPI_Send-MPI_Recv con Intel-MPI en el Superdome Ben

- Para *p* entre 8 y 80 :
  - *n* pequeños: varios picos irregulares
  - A partir de *n* determinado: comportamiento crecimiento lineal
- Para *p* > 80
  - Gran variabilidad de tiempos según *n*
  - En general, un importante incremento de tiempos respecto *p* pequeños

Uso de MPI en el supercomputador Ben Arabí
Comparación prestaciones con distintos entornos
Comunicaciones punto a punto



n	$p=8$	$p=16$	$p=32$	$p=48$	$p=64$	$p=80$	$p=96$	$p=112$	$p=128$
4	HP	HP	HP	HP	HP	HP	HP	HP	HP
	80 %	86 %	89 %	86 %	87 %	88 %	100 %	100 %	100 %
12	HP	HP	HP	HP	HP	HP	HP	HP	HP
	85 %	90 %	87 %	81 %	78 %	85 %	99 %	100 %	100 %
44	HP	HP	HP	HP	HP	HP	HP	HP	HP
	59 %	72 %	54 %	21 %	16 %	65 %	98 %	100 %	100 %
60	HP	HP	HP	Intel	Intel	HP	HP	HP	HP
	56 %	67 %	35 %	8 %	14 %	19 %	98 %	99 %	99 %
96	HP	HP	Intel	Intel	Intel	Intel	HP	HP	HP
	30 %	1 %	30 %	36 %	47 %	28 %	94 %	99 %	99 %
128	HP	Intel	Intel	Intel	Intel	Intel	HP	HP	HP
	27 %	7 %	49 %	54 %	49 %	41 %	97 %	94 %	99 %
228	Intel	Intel	Intel	Intel	Intel	Intel	HP	HP	HP
	5 %	56 %	67 %	67 %	64 %	64 %	95 %	96 %	97 %
512	Intel	Intel	Intel	Intel	Intel	Intel	HP	HP	HP
	61 %	66 %	69 %	64 %	70 %	70 %	87 %	95 %	80 %
1024	Intel	Intel	Intel	Intel	Intel	Intel	HP	HP	HP
	66 %	67 %	68 %	68 %	67 %	67 %	19 %	89 %	64 %
2048	Intel	Intel	Intel	Intel	Intel	Intel	HP	HP	HP
	63 %	62 %	66 %	65 %	65 %	65 %	48 %	68 %	37 %
4096	Intel	Intel	Intel	Intel	Intel	Intel	HP	Intel	HP
	64 %	66 %	65 %	64 %	63 %	64 %	37 %	10 %	49 %

Tabla 2: Selección de la biblioteca óptima en el HP Integrity Superdome para diferentes tamaños de mensaje (n) y número de procesos (p) para la operación punto a punto síncrona. Se muestra la diferencia en tanto por ciento entre los tiempos de envío cuando se selecciona la biblioteca con el mejor tiempo.

Uso de MPI en el supercomputador Ben Arabí
Comparación prestaciones con distintos entornos
Comunicaciones colectivas

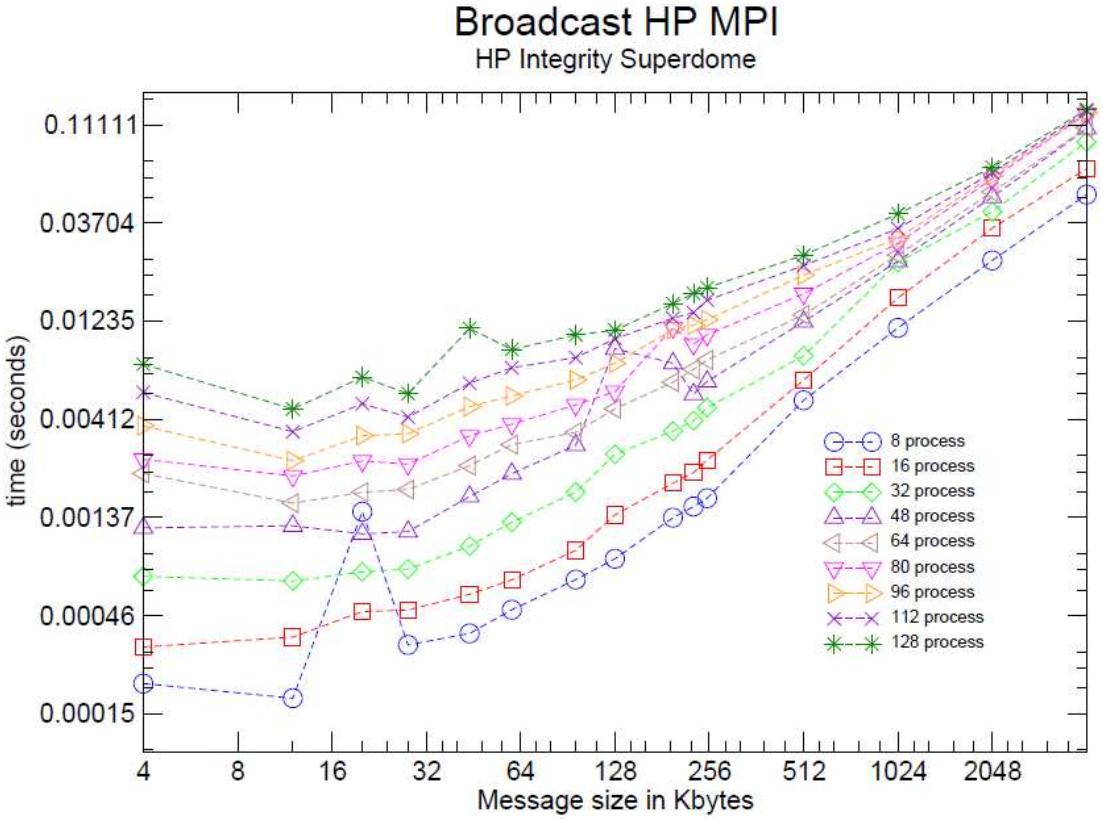


Figura 5: Colectiva Broadcast con HP-MPI para diferentes tamaños de mensaje en el HP Integrity Superdome.

Uso de MPI en el supercomputador Ben Arabí
Comparación prestaciones con distintos entornos
Comunicaciones colectivas

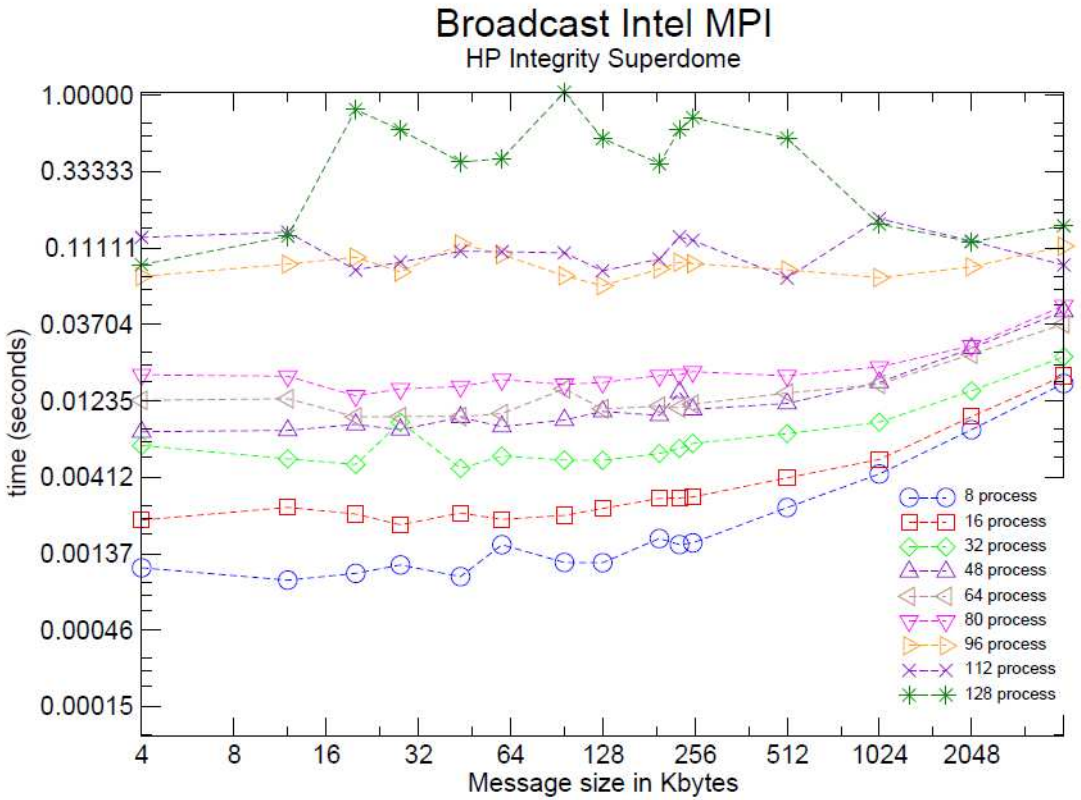
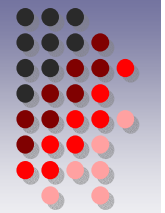


Figura 6: Colectiva Broadcast con Intel MPI para diferentes tamaños de mensaje en el HP Integrity Superdome.

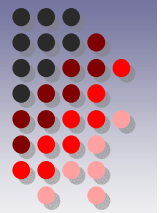
Uso de MPI en el supercomputador Ben Arabí
Comparación prestaciones con distintos entornos
Comunicaciones colectivas



n	$p=8$	$p=16$	$p=32$	$p=48$	$p=64$	$p=80$	$p=96$	$p=112$	$p=128$
4	HP	HP	HP	HP	HP	HP	HP	HP	HP
	81 %	88 %	87 %	86 %	81 %	90 %	97 %	95 %	94 %
28	HP	HP	HP	HP	HP	HP	HP	HP	HP
	61 %	78 %	83 %	88 %	82 %	83 %	95 %	96 %	99 %
96	HP	HP	HP	HP	HP	HP	HP	HP	HP
	42 %	85 %	69 %	67 %	80 %	70 %	93 %	92 %	99 %
252	Intel	HP	HP	HP	HP	HP	HP	HP	HP
	12 %	19 %	28 %	43 %	31 %	42 %	86 %	88 %	96 %
1024	Intel	Intel	Intel	Intel	Intel	Intel	HP	HP	HP
	63 %	65 %	61 %	30 %	38 %	30 %	65 %	58 %	63 %
4096	Intel	Intel	Intel	Intel	Intel	Intel	Intel	Intel	Intel
	69 %	72 %	75 %	59 %	64 %	66 %	14 %	17 %	9 %

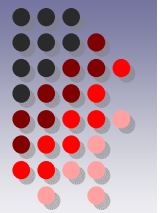
Tabla 3: Selección de la biblioteca óptima en el HP Integrity Superdome para diferentes tamaños de mensaje (n) y número de procesos (p) para la operación colectiva de broadcast. Se muestra la diferencia en tanto por ciento entre los tiempos de envío cuando se selecciona la biblioteca con el mejor tiempo.

Indice de la sesión



- Introducción
- Comunicaciones punto a punto
- Comunicaciones colectivas
- Agrupación de datos
- Comunicadores y topologías
- MPI-2
- Uso de MPI en el supercomputador BenArabi
- **Ejercicios prácticos**

Ejercicios



1.- Copia a tu directorio y prueba con el entorno de intel los ejemplos que están en: `/users/acuencam/ejemplos_mpi`

1.1.- Conectarse a `arabi.fpcmur.es`

```
ssh arabi.fpcmur.es
usuario          XXXX
clave            YYYY
```

1.2.- Para obtener los programas de ejemplo:

```
mkdir $HOME/ejemplos_mpi
cp /home/acuencam/ejemplos_mpi/*.c $HOME/ejemplos_mpi/.
```

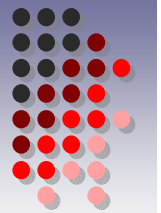
1.3.- Para cargar el modulo para ejecutar y compilar MPI de intel:

```
module unload pmpi
module load impi
```

1.4.- Para compilar el programa `hello.c` con `icc`:

```
mpiicc hello.c  -o hello_intel
```

Ejercicios



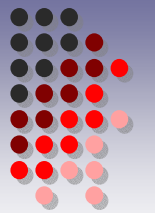
1.5.- Escribir este script “trabajo_hello_intel” :

```
#!/bin/bash
#BSUB -a intelmpi            # Se especifica el tipo de mpi
#BSUB -J hello_intel         # Nombre del trabajo
#BSUB -o hello_intel.%J.out  # Nombre del fichero de salida
#BSUB -e hello_intel.%J.err  # Nombre del fichero de error
#BSUB -q arabi_short         # Nombre de la cola
#BSUB -n 4                   # Núm de cores a reservar = num de
                  procesos

# Cargamos Entorno Necesario
source /etc/profile.d/modules.sh
module unload pmpi
module load impi

# Ejecución del trabajo
mpirun.lsf /users/$USER/ejemplos_mpi/hello_intel
```

1.6.- Enviar el trabajo “trabajo_hello_intel” a la cola:

```
bsub < trabajo_hello_intel
```

Ejercicios



2.- Copia a tu directorio y prueba con el entorno de hp (platform) los ejemplos que están en: `/users/acuencam/ejemplos_mpi`

2.1.- Conectarse a `arabi.fpcmur.es`

```
ssh arabi.fpcmur.es
usuario          XXXX
clave            YYYY
```

2.2.- Para obtener los programas de ejemplo:

```
mkdir $HOME/ejemplos_mpi
cp /home/acuencam/ejemplos_mpi/*.c $HOME/ejemplos_mpi/.
```

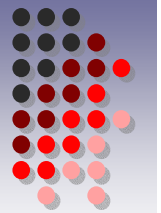
2.3.- Para cargar el modulo para ejecutar y compilar MPI de HP:

```
module unload impi
module load pmpi
```

2.4.- Para compilar el programa `hello.c` con `cc`:

```
mpicc hello.c  -o hello_hp
```

Ejercicios



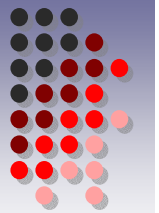
2.5.- Escribir este script “trabajo_hello_hp” :

```
#!/bin/bash

#BSUB -J hello_hp            # Nombre del trabajo
#BSUB -o hello_hp.%J.out     # Nombre del fichero de salida
#BSUB -e hello_hp.%J.err     # Nombre del fichero de error
#BSUB -q arabi_short         # Nombre de la cola
#BSUB -n 4                   # Númcores reservar = num procesos

# Cargamos Entorno Necesario
source /etc/profile.d/modules.sh
module unload impi
module load pmpi

# Ejecución del trabajo
mpirun –lsb_hosts /users/$USER/ejemplos_mpi/hello_hp
```

2.6.- Enviar el trabajo “trabajo_hello_hp” a la cola:

```
bsub < trabajo_hello_hp
```

Ejercicios



3. Amplia el programa broadcast.c de manera que además de los datos que se distribuyen en la versión actual también se envie una cadena de 10 caracteres.

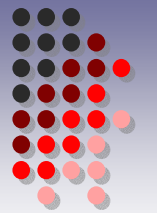
4. Modifica el programa gather.c para que además del vector **datos**, se recopile en el proceso raiz el vector **resultados**, también formado de enteros.

5. Modifica el programa scatter.c para que el array **datos** distribuido desde el proceso 0 contenga números reales de doble precisión en lugar de enteros.

6. Amplia el programa reduce.c para que, además de calcular la suma, calcule el valor máximo relativo a cada posición de los arrays locales de los distintos procesos (el máximo de la posición 0, el de la posición 1,…)

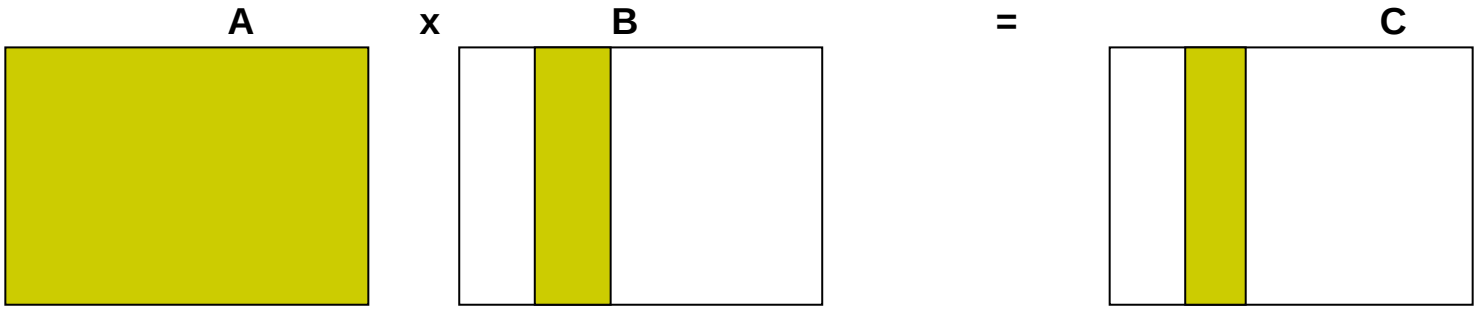
7. Amplia el programa hello.c para que al final del programa cada proceso **i** mande otro mensaje de saludo al nodo **(i+1)%p**, y tras ello, cada proceso muestre el mensaje recibido en pantalla.

Ejercicios



8. Completa el programa de ejemplo **mult_mpi.c** (calcula el producto de matrices C <= A*B) que conlleva:

- **broadcast**: La matriz A se manda completamente a todos.
- **scatter**: La matriz B se distribuye por bloques de columnas.
- **mm**: Cada proceso calcula un bloque de columnas de C multiplicando la matriz A por el bloque de columnas de B que le ha correspondido. Por ejemplo, P_1 calcularía:



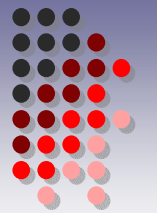
- **gather**: Todos los procesos mandan al root sus resultados parciales, es decir, sus bloques de columnas de C. Al finalizar, el proceso root contendrá la matriz C resultado completamente.

NOTA: Para la compilación usar el sript: `compilarblas_hp`, o bien, `compilarblas_intel`:

```
cp /users/acuencam/ejemplos_mpi/compilarblas* $HOME/ejemplos_mpi/.

compilarblas_hp mult_mpi
```

NOTA: Para la ejecución, añadir "`module load mkl`" al script del trabajo a mandar a la cola

Ejercicios



9. Amplia el programa **derivados.c** para que el tipo derivado que se forma y se envía contenga también una cadena de 48 caracteres.

10. Amplia el programa **empaquetados.c** para que el paquete de datos que se forma y se envía contenga también un vector de 100 números reales de doble precisión.