Arquitectura y Programación de Altas Prestaciones

Práctica 1: MPI

Especialidad Ingeniería de Computadores
Departamento de Arquitectura y Tecnología de los Computadores
Maribel García Arenas
mgarenas@ugr.es

Antes de empezar

- Enviar login personal a mgarenas@ugr.es
- Este login será puesto en un fichero de usuarios que tendrán permisos especiales para poder conectarse de forma remota a varios ordenadores de la misma aula.
- Ese fichero de permisos se leerá sólo si arrancáis la versión 10.04 de Ubuntu con el código MPI
- El fichero de permisos permitirá que se inserten en el fichero /etc/passwords todos los logins y passwords de todos los compañeros para poner acceder a ellos y a los ordenadores que tengan encendidos durante el horario de prácticas

Configuración de cuenta para ejecutar un shell remoto en Linux

- Consultar donde está montado tu home << echo $HOME>>
- Ir a dicho directorio
- Ejecutar una orden de forma remota:
  - ssh hostname orden
- Ejecutar de forma remota la orden pwd en el ordenador de tu compañero
- ¿Qué ocurre?
- ¿Qué es el fingerprint?

Par de claves privadas y públicas RSA

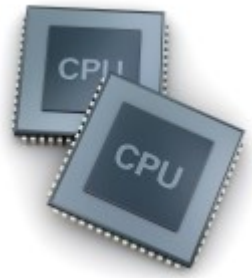
- Generar un par de claves dentro del directorio .ssh dentro de tu $HOME
  - ssh-keygen
- Aceptar las opciones por defecto.
- Examinar los ficheros generados e indicar qué contienen
- Autorizar mi clave publica para que sea una clave autorizada
  - Renombrar id_rsa.pub a authorized_keys
- Exportar clave publica a todos los ordenadores que queramos utilizar de forma remota (ssh-copy-id)
  - scp id_rsa.pub login@ordenador:/$HOME/.ssh
- El último paso tenéis que hacerlo aunque funcionaría también sin él, ¿por qué?

Copiar Material

- Crear dentro de nuestro $HOME la carpeta ACAP
- Crear directorio P1
- Bajar material de swad a P1
- Examinar ficheros copiados y realizar una relación y qué contiene cada uno de ellos
- Compilar y ejecutar cpi-seq.c
- ¿Tiene errores?
- ¿Lo habéis ejecutado bien a la primera?
- ¿Qué hace este código?

Programación Paralela

- Dentro de la computación paralela se engloban los siguientes aspectos:
  - Diseño de computadores paralelos
    - Escalables y con velocidad en las comunicaciones
  - Diseño de algoritmos paralelos eficientes
    - Minimicen las comunicaciones y sean portables
  - Métodos de evaluación de estos algoritmos
  - Lenguajes de programación
    - Flexibilidad y abstracción frente a las distintas arquitecturas
  - Herramientas de programación paralela
    - Depuración, simulación y visualización
  - Programación automática de computadores paralelos
    - Compiladores que paralelicen



Paradigma de programación con máquinas de memoria distribuida

- **Paso de mensajes**
- Es el paradigma más usado para la programación de multicomputadores de memoria distribuida.
- Consiste en que los procesos se comuniquen mediante en envío y la recepción explícita de mensajes.
- Dentro de este paradigma se han desarrollado varias tecnologías:
  - MPI (Message Passage Interface)
  - PVM (Parallel Virtual Machine)

Ventajas o desventajas

- **Principales ventajas de MPI sobre PVM**
- **MPI tiene más de una implementación de calidad de distribución gratuita.** Las implementaciones LAM, MPICH y OpenMPI son las más populares. La elección de las herramientas de desarrollo no está sujeta a la implementación escogida.
- **MPI tiene comunicaciones completamente asíncronas.** Los envios y las recepciones de los mensajes pueden solaparse completamente con la computación.
- **MPI posee grupos que son sólidos, eficientes y deterministas.** La pertenencia a un grupo es estática, esto implica que no hay condiciones de paso causadas por un proceso que independientemente acceda a un grupo o que salga, la formación de los grupos es colectiva.

Características MPI

- **MPI hace una sincronización que protege otro software en ejecución.** Las comunicaciones que se realizan entre grupos son marcadas de modo que no interfieran con los parámetros de otras comunicaciones que utilicen la librería.
- **MPI es completamente portable.**
- **MPI está formalmente especificada.** Todas las implementaciones tienen que cumplir las especificaciones.
- **MPI es un estándar.** Fue diseñada según un consenso abierto.
- Documentación en : http://www.open-mpi.org/doc/

Ejemplo 0, Analizar y explicar qué hace

```
#include <stdio.h>
#include <string.h>
#include “mpi.h”

main(int argc, char * argv[]){
    int mi_rank, p, source, dest, tag=0;
    char message[100];

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

    if(mi_rank!=0){
         sprintf(message, “Hola desde el procesador %d”, mi_rank);
         dest=0;
         MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    else{
         for (source=1; source < p; source++){
              MPI_Recv(message,100,MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
              printf(“%s \n”, message);
         }

    }
    MPI_Finalize();
}
```

MPI_INIT

MPI_Init – Inicializa el entorno de ejecución MPI

```
MPI_Init(&argc,&argv);

int MPI_Init(int *pargc, char ***pargv)
```

Parametros de entrada:

pargc
- Puntero al número de argumentos

pargv
- Puntero al vector de argumentos

Observaciones:

En la especificación de MPI no se utilizan los argumentos de la línea de comandos aunque también se da la oportunidad de usarlos. La implementación de LAM/MPI utiliza el comando de ejecución *mpirun* para obtener los parámetros ´de la línea de comandos.

MPI_Finalize

MPI_Finalize – Termina la ejecución del entorno MPI

```
MPI_Finalize();

int MPI_Finalize(void)
```

Observaciones:

Todos los procesos deben invocar esta función antes de terminar su ejecución y, es recomendable que sea la última línea en el código (antes de un return).

MPI_COMM_RANK

MPI_Comm_rank - Determina el indentificador (rank) del proceso

```
MPI_Comm_rank(MPI_COMM_WORLD, &mi_rank);

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

Entrada:

comm - comunicador

Salida:

rank - rank del proceso que ha invocado la función dentro del comunicador comm

ERRORS

Cuando ocurre un error se aborta la ejecución de este proceso.

MPI_COMM_SIZE

MPI_Comm_size – Determina el tamaño del grupo asociado al comunicador

```
MPI_Comm_size(MPI_COMM_WORLD, &p);

int MPI_Comm_size(MPI_Comm comm, int *psize)

Entrada:
        comm    - comunicador
Salida:
        psize   - número de procesos dentro del grupo del
    comunicador.

Observaciones:

        MPI_COMM_NULL is not considered a valid argument to this
    function.
```


Comunicación proceso 0

```
if(mi_rank!=0){
    sprintf(message, “Hola desde el procesador %d”, mi_rank);
    dest=0;
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag,
    MPI_COMM_WORLD);
}
else{
    for (source=1; source < p; source++){
       MPI_Recv(message,100,MPI_CHAR, source, tag, MPI_COMM_WORLD,
        &status);
       printf(“%s \n”, message);
    }
```

Si el identificador del proceso es distinto de 0, envío mi mensaje al proceso 0, si sí soy el proceso 0, recibo todos los mensajes enviados por el resto de los procesos y los imprimo en pantalla

Un elemento importante a considerar es que proceso puede hacer las operaciones de entrada salida sobre la entrada y salida estándar. Depende de la configuración del sistema.

Comunicaciones 1-1:MPI_SEND

MPI_Send - Realiza un envío básico

```
MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag,
   MPI_COMM_WORLD);
```

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

Entrada:

buf	-	dirección de inicio del buffer a enviar
count	-	número de elementos a enviar
dtyp	-	tipo de dato de MPI que se va a enviar
dest	-	rank del proceso de destino
tag	-	etiqueta
comm	-	comunicador

Observaciones:

Esta función realiza un envio bloqueante hasta que el mensaje sea totalmente recibido por el destinatario.

Comunicaciones 1-1:MPI_RECV

MPI_Recv - Recepción básica

```
MPI_Recv(message,100,MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);

int MPI_Recv(void *buf, int count, MPI_Datatype dtype,
                  int src, int tag, MPI_Comm comm, MPI_Status *stat)

Entrada:
       count  - máximo número de elementos a recibir
       dtype  - tipo de dato a recibir
       src    - rank del origen del mensaje
       tag    - etiqueta
       comm   - comunicador

Salida:
       buf    - dirección de memoria de comienzo del buffer
       stat   - objeto Status, que puede ser sustituido por la constante de MPI:
    MPI_STATUS_IGNORE si los datos de estado no interesan.

Observaciones:

    para saber exactamente el número de elementos recibidos hay que mirar el estado de la
    recepción mediante MPI_Get_count.

    Dentro de las constantes definidas en MPI, tenemos:
    MPI_ANY_SOURCE Indica que el mensaje a recibir puede venir de cualquier destinatario
    MPI_ANY_TAG Indica que la etiqueta no tiene que ser igual a la del envío
```


MPI_Get_COUNT

MPI_Get_count - calcula el número de elementos recibidos

```
int MPI_Get_count(MPI_Status *stat, MPI_Datatype dtype,
                  int *count)
```

Entrada:

- stat - el Status de la recepción
- dtype - el tipo de dato recibido en el buffer

Salida:

- count - número de elementos recibidos

Observaciones:

No es posible pasarle MPI_STATUS_IGNORE.

MPI_Status

Este tipo de dato es una estructura que contiene los siguientes elementos que pueden ser utilizados por el programador:

MPI_SOURCE	Indica el rank del proceso que envió el mensaje
MPI_TAG	Indica la etiqueta del mensaje
MPI_ERROR	Indica el posible error

Comunicaciones 1-M

- Se realizan de forma síncrona, todos los procesos la realizan al mismo tiempo.
- Están implementadas para simplificar y optimizar las comunicaciones entre varios procesos.
- Para sincronizar a los procesos sin tener que realizar ninguna operación disponemos de la función MPI_Barrier(comm).

MPI_BCast

MPI_Bcast - Envía un mensaje desde el proceso con rank "root" a todos los demás procesos en el comunicador

int MPI_Bcast(void *buff, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Entradas/salida:

buff - dirección de inicio del buffer
count - número de elementos en el buffer
datatype – tipo de dato de los elementos del buffer
root - rank del proceso que envía
comm - comunicador donde se envía el mensaje

Observaciones:

Si hay menos de 4 procesos se itera sobre un bucle, en caso contrario, se utiliza un algoritmo basado en una estructura de árbol.

MPI_Reduce

MPI_Reduce - Reduce una serie de valores a un único valor en el procesador root

int MPI_Reduce(void *sbuf, void* rbuf, int count, MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm)

Entradas:

sbuf - dirección del buffer donde están los datos a reducir
rbuf - dirección del buffer donde se almacena el resultado
count - número de elementos en el buffer
dtype - tipo de dato
op - operación de reducción
root - rank del proceso “root”
comm - comunicador

Salidas

rbuf - `dirección de memoria de comienzo del buffer (root)`

Observaciones:

Si hay menos de 4 procesos, se itera sobre un bucle normal recibiendo de los procesos, si no, se emplea una estructura en árbol donde se aplica la reducción en cada nodo padre, distribuyendo de este modo el cálculo de la operación.

MPI_Reduce II

- Tipos de operaciones de reducción:
  - MPI_MAX maximum
  - MPI_MIN minimum
  - MPI_SUM sum
  - MPI_PROD product
  - MPI_LAND logical and
  - MPI_BAND bit-wise and
  - MPI_LOR logical or
  - MPI_BOR bit-wise or
  - MPI_LXOR logical xor
  - MPI_BXOR bit-wise xor
  - MPI_MAXLOC max value and location
  - MPI_MINLOC minimum value and location
- Se pueden definir operaciones específicas de reducción

MPI_AllReduce

MPI_Allreduce - Reduce una serie de valores a un único valor en todos los procesadores

int MPI_AllReduce(void *sbuf, void* rbuf, int count, MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm)

Entradas:

sbuf - dirección del buffer donde están los datos a reducir
count - número de elementos en el buffer
dtype - tipo de dato
op - operación de reducción
root - rank del proceso “root”
comm - comunicador

Salidas

rbuf - `dirección de memoria de comienzo del buffer (root)`

Observaciones:

Es equivalente a realizar un MPI_Reduce y luego un MPI_Bcast. Se hace una comunicación estructurada en árbol para cada proceso.

Ejemplo MPI_Reduce

```
if (my_rank == 0) {
        total = integral;
        for (source = 1; source < p; source++) {
            MPI_Recv(&integral, 1, MPI_DOUBLE, source, tag, MPI_COMM_WORLD, &status);
            total = total + integral;
        }
    }
   else {
        MPI_Send(&integral, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
    }
```

```
int MPI_Reduce(void *sbuf, void* rbuf, int count, MPI_Datatype dtype, MPI_Op op, int root,
      MPI_Comm comm)
```

D);

MPI_Gather, MPI_ALLGather

MPI_Gather - Reune valores de un grupo de procesos en un proceso
MPI_Allgather - Reune valores de un grupo de procesos en todos los procesos

```
int MPI_Gather(void *sbuf, int scount, MPI_Datatype sdtype,
               void *rbuf, int rcount, MPI_Datatype rdtype,
               int root, MPI_Comm comm)
int MPI_Allgather(void *sbuf, int scount, MPI_Datatype sdtype,
               void *rbuf, int rcount, MPI_Datatype rdtype,
               int root, MPI_Comm comm)
```

Entradas:

- sbuf - dirección del buffer con los datos a enviar
- scount – numero de elementos a enviar
- sdtype – tipo de datos
- rcount – numero de elementos a recibir por proceso(no el total recibido)
- rdtype - tipo de datos
- root - rank del root
- comm - comunicador

Salidas:

- rbuf - dirección del buffer con los datos recibidos

MPI_Scatter

MPI_Scatter - Dispersa datos desde un proceso a todos los demás en el grupo

```
int MPI_Scatter(void *sbuf, int scount, MPI_Datatype sdtype,
                void *rbuf, int rcount, MPI_Datatype rdtype,
                int root, MPI_Comm comm)
```

Entradas:

- sbuf - dirección del buffer con los elementos a enviar
- scount – número de elementos a enviar
- sdtype – tipo de datos
- rcount – número de elementos a recibir
- rdtype – tipo
- root - del que envía
- comm - comunicador

Salidas:

- rbuf - dirección del buffer de almacenamiento

MPI_Reduce_scatter

MPI_Reduce_scatter

Primero realiza una reducción en un vector distribuido en los procesos y luego reparte el vector entre los procesos.

MPI_Reduce_scatter (&sendbuf,&recvbuf,recvcount,datatype,
..... op,comm)

MPI_Alltoall

Cada proceso en el grupo realiza una dispersión (Allscatter) de sus datos sobre el resto de procesos.

MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf,
..... recvcnt,recvtype,comm)

Estructuras de datos

Tipos de datos derivados y empaquetamiento de datos

- En una operación de envío, si tenemos que enviar tres datos independientes, tenemos que realizar 3 operaciones diferentes, esto se puede optimizar enviando en el mismo mensaje los datos.
- Problema: los datos a enviar no están en posiciones de memoria continuas desde la dirección de comienzo del buffer.
  Solución: Definir tipos de datos derivados o empaquetar datos.

Estructuras de datos II

- MPI permite la definición de estructuras de datos complejas (tipos de datos derivados) a partir de los tipos de datos simples que proporciona. De este modo se pueden enviar mensajes que contengan varios tipos de datos.
- Gracias a los tipos derivados podremos enviar mensajes cuyo contenido no es contiguo en memoria.
- Un tipo de dato derivado consiste en pares (tipo de dato, desplazamiento) donde tipo de dato es un dato del tipo MPI_xxx y desplazamiento es el número de bytes que hay entre la posición de memoria del inicio del buffer y el comienzo del dato.
- MPI proporciona varios modos para la creación de tipos de datos derivados: contiguos, vectores, indexados y estructurados.

MPI_Type_STRUCT

MPI_Type_struct(n_elem,block_lengths,displacements, typelist, tipo_nuevo);

n_elem: número de bloques en la estructura (int)
block_lengths: número de elementos en cada bloque (int [])
displacements: desplazamiento de cada bloque desde el primer elemento (MPI_Aint [])
typelist: tipos de dato MPI de los bloques (MPI_Datatype [])
tipo_nuevo: nuevo tipo de dato (MPI_Datatype *) (por ref.)

MPI_Type_commit(MPI_Datatype * tipo_nuevo)

confirma su uso para la comunicación (podría usarse únicamente para formar otros tipos de datos más complejos).

MPI_Address(void * elemento, MPI_Aint * direc)

almacena en direc la dirección de memoria del elemento
es igual que “direccion = &variable;”, pero así se asegura la portabilidad

Estructuras de datos

- El resto de tipos derivados hacen referencia a conjuntos de elementos del mismo tipo pero distribuidos de una forma no continua en memoria.
- Es especialmente útil a la hora de realizar operaciones con secciones de matrices.
- A la hora de hacer comunicaciones entre pares de procesos, se pueden enviar tipos de datos derivados sin necesidad de haber creado el tipo en el proceso receptor, bastaría con indicar el número de elementos del tipo a recibir (type matching)

MPI_Type_FREE

- Una vez usado un tipo de dato derivado puede ser necesario liberar memoria, para ello utilizaremos: **MPI_Type_free(MPI_Datatype * datatype)**

Tipos Derivados: Ejemplo

```
float a,b;
int n, long_bloque[3];
MPI_Aint desplazamientos[3];
MPI_Datatype lista_de_tipos[3];

/* Para los cálculos de direcciones */
MPI_Aint start_address;
MPI_Aint address;

/* Nuevo tipo */
MPI_Datatype nuevo_tipo;
/* Nuestros datos son de un solo elemento cada uno */
long_bloque[0] = long_bloque[1] = long_bloque[2] = 1;

/* Nuestras variables son dos flotantes y un entero */
lista_de_tipos[0] = lista_de_tipos[1] = MPI_FLOAT;
lista_de_tipos[2] = MPI_INT;

/* El primer elemento a lo ponemos a desplazamiento 0 */
desplazamientos[0] = 0;

/* Calculamos los otros desplazamientos respecto de a */
MPI_Address(&a, &start_address);
MPI_Address(&b, &address);
desplazamientos[1] = address - start_address;
MPI_Address(&n, &address);
desplazamientos[2] = address - start_address;

/* Construimos el tipo derivado */
MPI_Type_struct(3, long_bloque, desplazamientos, lista_de_tipos, &nuevo_tipo);

/* Informamos al sistema del nuevo tipo de datos */
MPI_Type_commit(&nuevo_tipo);
```

Empaquetamiento de datos

- Podemos crear explícitamente un buffer con los datos que queremos enviar, siempre que estén almacenados en zonas de memoria contiguas.
- Mediante la función MPI_Pack iremos concatenando los elementos de distintos tipos:
- **MPI_Pack(void * elem,int n_elem, MPI_Datatype tipo, void * buffer,int buffer_size,int * posicion, MPI_Comm comunicador)**

- Una vez enviado el buffer, los datos se recuperan en el mismo orden en el que se introdujeron mediante:
- **MPI_Unpack(void * buffer,int size, int * posicion, void * elem, int n_elem,MPI_Datatype tipo, MPI_Comm comunicador)**

Empaquetamiento de datos


```
#define TAM_BUFFER 100
#define TAG 0
#define DEST 1
#define SOURCE 0

main(int argc, char* argv[]) {
  int   my_rank, posicion, n;
  char  buffer[TAM_BUFFER];
  MPI_Status  status;
  float a, b;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   if ( rank == SOURCE ) {
    a = 10.3;       b = 11.2      n = 4;   posicion = 0;
    MPI_Pack(&a,1,MPI_FLOAT,buffer,TAM_BUFFER,&posicion,MPI_COMM_WORLD);
    MPI_Pack(&b,1,MPI_FLOAT,buffer,TAM_BUFFER,&posicion,MPI_COMM_WORLD);
    MPI_Pack(&n,1,MPI_INT,buffer,TAM_BUFFER,&posicion,MPI_COMM_WORLD);
    MPI_Send(buffer,TAM_BUFFER,MPI_PACKED,DEST,TAG,MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(buffer,TAM_BUFFER,MPI_PACKED,SOURCE,TAG,MPI_COMM_WORLD,&status);
    posicion = 0;
    MPI_Unpack(buffer,TAM_BUFFER,&posicion,&a,1,MPI_FLOAT,MPI_COMM_WORLD);
    printf("   a = %f\n",a);
    MPI_Unpack(buffer,TAM_BUFFER,&posicion,&b,1,MPI_FLOAT,MPI_COMM_WORLD);
    printf("   b = %f\n",b);
    MPI_Unpack(buffer,TAM_BUFFER,&posicion,&n,1,MPI_INT,MPI_COMM_WORLD);
    printf("   n = %d\n",n);
  }
  MPI_Finalize();
}
```

Comunicadores y Grupos

- Un grupo es un conjunto de procesos.
- Cada proceso dentro de un grupo tiene asociado un único identificador dentro de ese grupo.
- Los identificadores empiezan con el número 0 hasta N-1 donde N es el número de procesos del grupo.
- Los grupos se representan como objetos ocultos al programador que accede a ellos mediante manejadores (handles).
- Un grupo está siempre asociado con un comunicador.
- Cada mensaje debe especificar un comunicador. Al igual que con los grupos, los comunicadores están ocultos al programador.
- Desde la perspectiva del programador, no hay diferencia entre un grupo y un comunicador, si bien cabe la posibilidad de que se hayan creado dos comunicadores distintos utilizando el mismo grupo.

Comunicadores y Grupos

- Los motivos para usar comunicadores y grupos son:
  - Permitir la organización de los procesos.
  - Permitir las comunicaciones colectivas en subconjuntos de procesos.
  - Proporcionan la base para la definición de topologías virtuales.
  - Proporcionan comunicaciones seguras.

- Los grupos y los comunicadores pueden ser creados y destruidos dinámicamente durante la ejecución del programa.
- Cada proceso puede pertenecer a varios grupos y comunicadores teniendo distintos identificadores asignados.

Comunicadores y Grupos

- El procedimiento más común a la hora de usar grupos y comunicadores es:
  - Obtener el grupo asociado al comunicador global MPI_COMM_WORLD utilizando la función **MPI_Comm_group**.
  - Crear un nuevo grupo utilizando los identificadores del grupo original usando **MPI_Group_incl**
  - Crear un nuevo comunicador asociado al grupo nuevo usando **MPI_Comm_create**
  - Obtener el nuevo rank usando MPI_Comm_rank
  - Realizar las comunicaciones
  - Liberar recursos usando **MPI_Comm_free** y **MPI_Group_free**

- Cuando se crea un comunicador se realiza una sincronización al igual que con las comunicaciones colectivas.
- Podemos crear varios comunicadores simultáneamente mediante **MPI_Comm_split**, utilizando un identificador de grupo y otro de rango.
- Ejemplos en http://geco.mines.edu/workshop/class2/examples/mpi/c_ex10.c

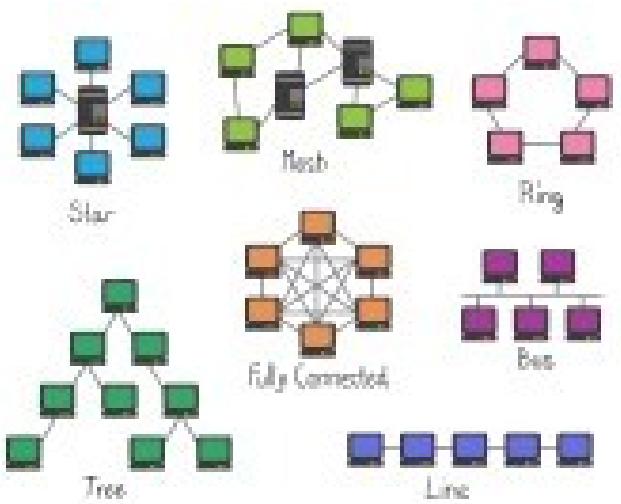
Comunicadores y Grupos: Ejemplo


```
/* Obtenemos el manejador del grupo global */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

/* Dividimos las tareas en dos conjuntos */
if (rank < NPROCS/2) {
  MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
  }
else {
  MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
  }

/* Creamos los comunicadores y realizamos las comunicaciones */
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

MPI_Group_rank (new_group, &new_rank);
printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);
```

Topologías Virtuales

- Son mecanismos para asociar diferentes modos de direccionamiento a la hora de realizar comunicaciones dentro de un grupo.
- Hay dos grupos: cartesianas (grid) y grafos.
- Las topologías son virtuales, es decir, la ordenación de los procesos no tiene nada que ver con la arquitectura real sobre la que se ejecuta el programa.
- Son útiles por:
  - Su utilidad en aplicaciones que siguen ciertos patrones de comunicación.
  - Pueden mejorar las prestaciones si es posible combinar la topología virtual con la topología real de la plataforma paralela.

Topologías Virtuales




```
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods,
    reorder, &cartcomm);

MPI_Comm_rank(cartcomm, &rank);

MPI_Cart_coords(cartcomm, rank, 2, coords);

MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP],
    &nbrs[DOWN]);
```

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

```
i n t dims [ 2 ] = { 3 , 2 } ;
i n t periods [ 2 ] = { false , true } ;
int reorder = false ;
MPI_Comm comm_2d ;
...
MPI_Cart_create (MPI_COMM_WORLD, 2 ,
dims , p e r i o d s , r e o r d e r , &comm_2d ) ;
```

Comunicaciones 1-1 Asíncronas

- Comunicaciones entre pares de procesos:
  - Bloqueantes
    - MPI_Send, MPI_Recv, MPI_Sendrecv, MPI_Wait, MPI_Waitall…
  - No bloqueantes
    - MPI_Isend, MPI_Irecv, MPI_Test, MPI_Testall, MPI_Iprobe

Prestaciones

- Normalmente se utilizan las siguientes medidas:
  - Tiempo de ejecución (respuesta) de una aplicación en el sistema. Se utiliza más en computación de altas prestaciones.
  - Productividad dada por el número de aplicaciones que se pueden procesar por unidad de tiempo. Se utiliza más en el ámbito de servidores.
  - Ganancia en velocidad Tiempo(1)/Tiempo(N)

Prestaciones

- Otras posibles medidas son:
  - Funcionalidad que se desee.
  - Alta disponibilidad, hace referencia a la existencia de elementos redundantes para reducir la degradación de prestaciones ante fallos.
  - Fiabilidad, tiempo que puede funcionar sin fallos (1=100%).
  - Tolerancia a fallos, requiere una alta disponibilidad
  - Expansibilidad, permite expandir el sistema modularmente.
  - Escalabilidad, es decir, que al aumentar los recursos, aumenten las prestaciones.

    Consumo de potencia.

Toma de medidas

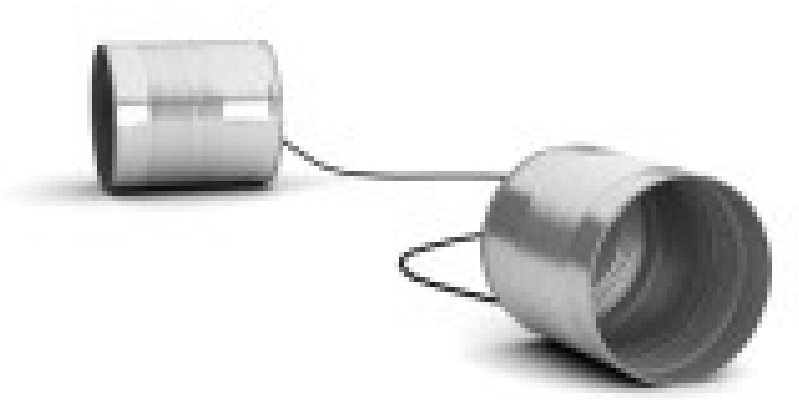
- El tiempo de ejecución paralelo viene dado por el tiempo de ejecución del programa más el tiempo que está parado y el tiempo de sobrecarga, To (overhead), que tiene como origen:
  - Tiempo para comunicación/sincronización
  - Tiempo para crear/terminar procesos
  - Tiempo de ejecución de las instru añadidas en la versión paralela

Ganancia en velocidad

- Idealmente, si tenemos p procesadores, el tiempo secuencial debería ser reducido a Ts/p, obteniendo una ganancia lineal.
- Esto normalmente está limitado al aprovechamiento del grado de paralelismo (número de tareas que se pueden ejecutar) y a la reducción debido a la sobrecarga.
- Es posible obtener ganancias superlineales:
  - Se introducen más recursos.
  - Dependiendo del problema, se explora más el espacio solución.

Tipos de comunicaciones

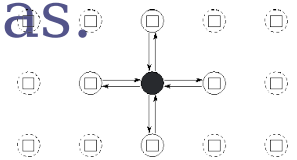
- En momento u otro de la ejecución, las tareas deberán comunicarse para transmitir datos (si se hizo una descomposición del dominio) de entrada o datos procesados (si se hizo una descomposición funcional).
- Es deseable que las operaciones de comunicación se hagan de forma distribuida y, por tanto, puedan ser simultaneas.
- Dentro de la metodología de diseño, se pueden distinguir las siguientes clases de comunicaciones:
  - Local
  - Global
  - Estructurada
  - No estructurada
  - Estática
  - Dinámica
  - Síncrona
  - Asíncrona



Tipos de comunicaciones

Local

- Cuando una tarea necesita los datos de un subconjunto de tareas, bien por solapamiento de datos a procesar o por la necesidad de utilizar los datos producidos por las tareas vecinas.



Global

- Este tipo de comunicación ocurre cuando muchas tareas participan de modo que es difícil agruparlas en zonas localizadas.
- Cuando se emplea este tipo de comunicación los problemas de centralización y secuencialidad pueden presentarse.

Agrupación

- Una vez descompuesto el problema, se puede optimizar su ejecución para un computador que posea menos procesadores que tareas definidas en los pasos anteriores.
- En esta fase se analiza si es recomendable replicar datos y/o operaciones o si es apropiado unir tareas (aglomerarlas).
- Los objetivos a cumplir son:
  - Reducir los costes de comunicación (incrementando la granularidad)
  - Mantener flexibilidad para escalar y mapear posteriormente

Mapeo

- Tiene como objetivo minimizar el tiempo de ejecución especificando donde se ejecutará cada tarea. Para ello:
  - Se mapean las tareas que ejecutan operaciones concurrentemente en procesadores distintos.
  - Se mapean tareas que se comunican frecuentemente (p.ej. Productor/consumidor) en el mismo procesador.
- Cuando también se planifica el orden de ejecución de las tareas dentro de un procesador también se denomina **planificación**.