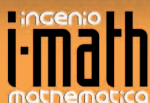
CURSO

INTRODUCCIÓN A LA PROGRAMACIÓN EN MPI

Organizado por:



Colaboran:

Grupo de Arquitectura de Computadores - USC

Red Mathematica Consulting & Computing de Galicia

Cofinanciado por:

Introducción a la programación en MPI

José Carlos Cabaleiro Domínguez
Xulio López Albín
Grupo de Arquitectura de Computadores
Dpto. Electrónica y Computación
Universidad de Santiago de Compostela

Índice

- Introducción
- MPI: conceptos generales
- Funciones de MPI básicas
- Comunicaciones colectivas
- Tipos de datos derivados
- Comunicadores
- Topologías
- Otros modos de comunicación
- Depurado de programas
- Introducción a MPI-2

Índice

- **Introducción**
- MPI: conceptos generales
- Funciones de MPI básicas
- Comunicaciones colectivas
- Tipos de datos derivados
- Comunicadores
- Topologías
- Otros modos de comunicación
- Depurado de programas
- Introducción a MPI-2

Introducción

- MPI (*Message Passing Interface*) es el estándar actual de programación de los sistemas de memoria distribuida, mediante **pase de mensajes**, (1993)
- Además, sirve para memoria compartida, clusters, redes de computadores, grid, etc.
- Sistemas heterogéneos
- Definida para C, C++, Fortran
- Se busca: portabilidad, eficiencia, …

Modelos de programación

- Pase de mensajes
- Data parallel
- Paralelización automática
- Memoria compartida: primitivas (*pthreads*)
- OpenMP
- Global arrays

Pase de mensajes

- Multiprocesadores de memoria distribuida y redes de computadores (sistemas heterogéneos)
  - Variables privadas
- Primitivas para:
  - Acceso a la memoria local de otros procesadores (comunicaciones).
  - Sincronización.
- Flexible: Control absoluto del programa
- Inconveniente: El programador es responsable la optimización del programa
  - Distribución de los datos y las computaciones, sincronización, comunicaciones, ...
- Normalmente, mayor rendimiento

Pase de mensajes

- Librerías estándar de pase de mensajes:
  - PARMACS, PVM (*Parallel Virtual machine*) y **MPI** (*Message Passing Interface*)
  - Existen librerías no estándar, desarrolladas para un multiprocesador determinado.
- Lenguaje C o Fortran con rutinas de comunicaciones

Data parallel

- Simplifica la programación con pase de mensajes.
  - Código secuencial con espacio de direcciones global.
  - Lenguaje estándar (C o Fortran)
  - Se genera código paralelo mediante directivas de compilación (p.e distribución de los datos)
- El compilador genera el código paralelo mediante
  - Distribución y partición de los datos (especificado por el programador)
  - Distribuyendo computaciones.
  - Insertando comunicaciones por pase de mensajes o memoria compartida.

Data parallel

- Características
  - Paralelismo a través de operaciones sobre matrices de datos
  - Directivas del compilador
- Detalles de bajo nivel transferidos al compilador
  - Eficiencia: mayor dependencia del compilador
- Objetivo:
  - Un mayor uso de los sistemas paralelos

Data parallel

- Fáciles de programar.
- Prueba con varias distribuciones de datos.
- Comunicaciones basadas en primitivas nativas del sistema ==> buen rendimiento
- Lenguajes:
  - CM Fortran, Vienna Fortran .
  - HPF (*High Performance Fortran*). Es el estándar.

Paralelización automática

- El compilador genera de forma automática la versión paralela de un código secuencial
- Herramientas:
  - Parafrase, SuperB, SUIF, Polaris, ...
- Campo puntero en la investigación
- Problemas con códigos irregulares y/o dinámicos

Memoria compartida

- Mecanismo de bajo nivel: *pthreads*.
- Primitivas:

pthread_create	pthread_cond_init
pthread_exit	pthread_cond_destroy
pthread_join	pthread_cond_wait
pthread_mutex_init	pthread_cond_signal
pthread_mutex_destroy	pthread_cond_broadcast

Memoria compartida: OpenMP

- Standard de programación en memoria compartida.
- Incluye directivas (#pragmas) de compilación junto con funciones de librería.
- Se basa en la semántica de *fork-join*.
- En una plataforma secuencial, las directivas son ignoradas (comentarios).

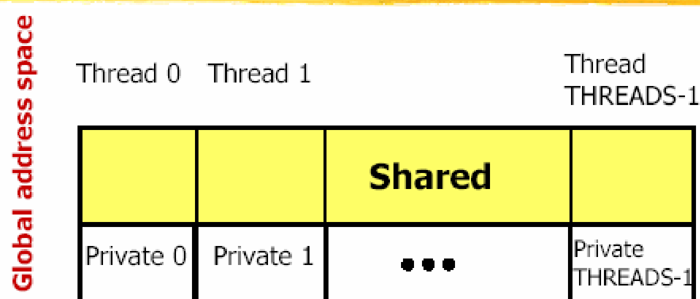
Comparativa

- Memoria compartida:
  - Más sencilla en su programación.
  - Más difícil cometer errores.
  - Las estructuras de datos cambian.
  - Se puede paralelizar incrementalmente.
- Pase de mensajes:
  - Más fácil obtener altos rendimientos.
  - Mayor portabilidad.
  - Dificultad en detectar y corregir errores.

Global arrays

- Modelo de programación de espacio de direcciones global particionado (*Partitioned global address space programming model*)
  - Espacio de direcciones lógicamente particionado
    - Memoria local y memoria remota
    - Permite datos compartidos y privados
    - Espacio compartido, lógicamente distribuido entre *threads*
    - Explota paralelismo en constelaciones (clusters de SMPs)
  - Equilibrio entre
    - Nivel de abstracción para simplificar la programación
    - Control directo para obtener rendimiento en arquitecturas actuales

Global arrays



- Ejemplos
  - UPC: *Unified Parallel C*
  - CAF: *Co-Array Frortran*
  - Titanium: Basado en Java o C++

Índice

- Introducción
- **MPI: conceptos generales**
- Funciones de MPI básicas
- Comunicaciones colectivas
- Tipos de datos derivados
- Comunicadores
- Topologías
- Otros modos de comunicación
- Depurado de programas
- Introducción a MPI-2

MPI: introducción

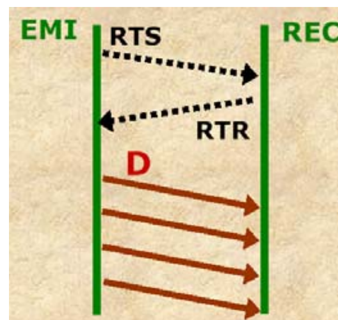
- MPI, es una librería de funciones de comunicación para el envío y recepción de mensajes entre procesos
- Definida para C, C++, Fortran y Fortran 90
- Necesario explicitar la comunicación entre procesos
  - Movimiento de datos
  - Sincronización

MPI: introducción

- Mensajes: deben incluir
  - Proceso que envía el mensaje, datos que se mandan, tipo de datos, número de datos, destinatario/s del mensaje, variable que recibe los datos, etc.
- Dos tipos de comunicaciones
  - Punto a punto
    - Involucran sólo a dos procesos (emisor-receptor), deben pertenecer al mismo *comunicador*
  - Colectivas
    - Son rutinas de comunicación que involucran a más de dos procesadores a la vez.
    - Pueden construirse a partir de comunicaciones punto a punto.
    - En MPI, deben pertenecer al mismo *comunicador*

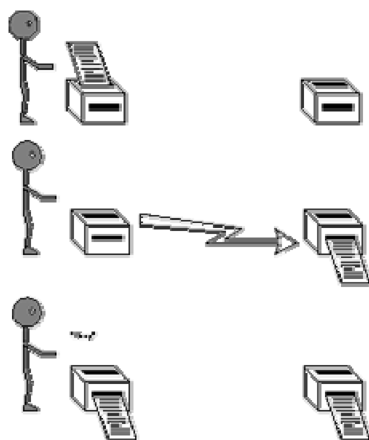
MPI: introducción

- Estrategias de comunicación
  - Síncrona



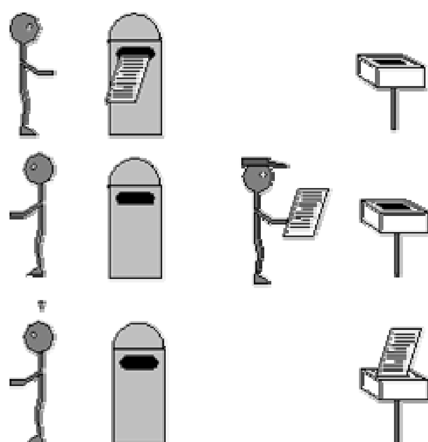
- Petición de transmisión (espera)
- Aceptación de transmisión
- Envío de datos (de buffer de usuario a buffer de usuario)

Envío síncrono



- El emisor necesita información sobre la recepción del mensaje
- La comunicación no se completa hasta que el mensaje ha sido recibido

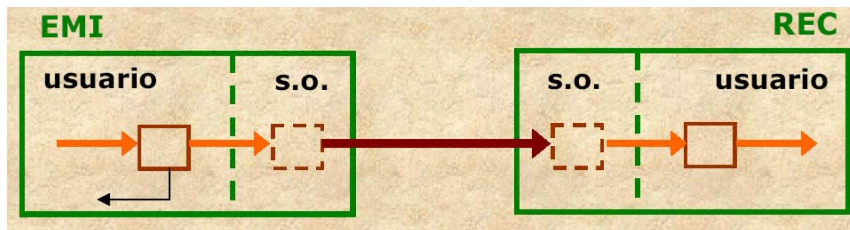
Envío asíncrono



- El emisor sólo conoce que envió el mensaje
- La comunicación se completa tan pronto como el mensaje se envía

MPI: introducción

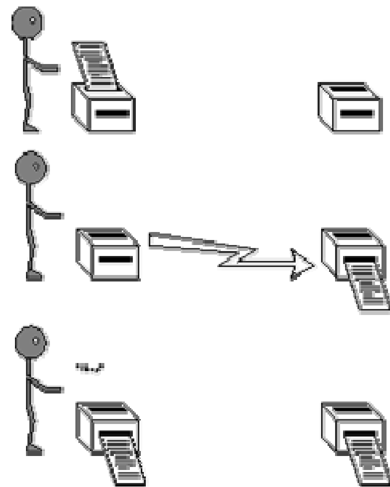
- Estrategias de comunicación
  - Buffered
    - Se realiza una copia del mensaje



MPI: introducción

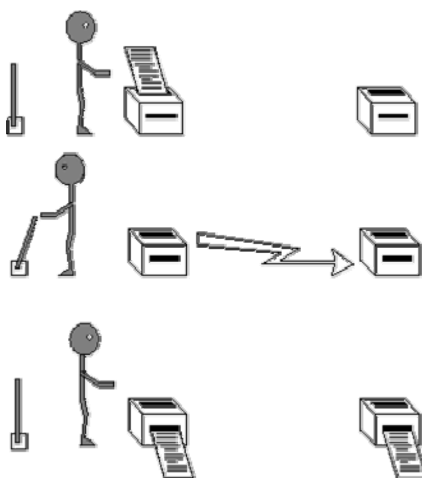
- Estrategias de comunicación
  - Bloqueante
    - Se espera a que la comunicación se produzca (o a que el buffer de usuario esté nuevamente disponible)
    - Se continúa la ejecución (usando una comunicación no bloqueante) y se comprueba más tarde si se ha producido
      - No bloqueante + función de espera = bloqueante
    - Síncrona es siempre bloqueante
- Buffered: ambas versiones

Operación bloqueante



- La subrutina de comunicación sólo acaba cuando la operación de comunicación se ha completado

Operación no bloqueante

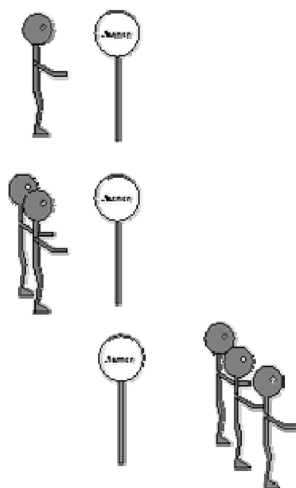


- Se inicia el proceso de envío del mensaje y se continúa el programa
- Existen funciones que chequean la recepción o esperan a que se reciba el mensaje

MPI: introducción

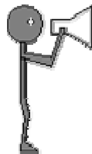
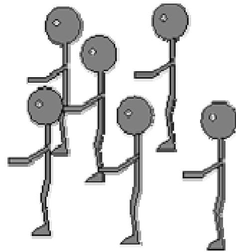
- Cada estrategia tiene sus ventajas e inconvenientes
  - Síncrona: más rápida si el receptor está preparado para recibir
    - Se ahorra la copia en el buffer
    - Posibilidad de *deadlock* (ya que es bloqueante)
  - Buffered: el emisor no se bloquea si el receptor no está disponible
    - Hay que hacer copia del mensaje
- La eficiencia de las comunicaciones determinará el rendimiento

Comunicaciones colectivas



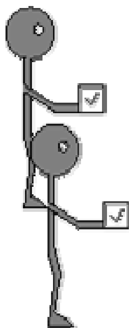
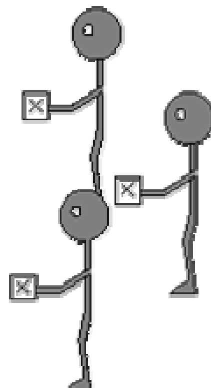
- **Barreras**
  - Sincronizan procesos
  - No hay intercambio de datos
  - Se detiene el programa hasta que todos los procesadores llegan a la barrera

Comunicaciones colectivas

- **Radiación**
  - Comunicación de uno a todos
  - Uno de los procesos envía un mensaje a varios destinatarios

Comunicaciones colectivas

- **Reducción**
  - Toma datos de diferentes procesadores y obtiene un único dato
  - El resultado puede ser radiado al resto de los procesos

MPI: introducción

- MPI implementa el modelo SPMD (*Single Program Multiple Data*)

```
if (pid == 1) ENVIAR_a_pid2
else if (pid == 2) RECIBIR_de_pid1
```

  - Espacio de direcciones privado
- MPI asume gestión estática de procesos (número y asignación)
  - Cada proceso tiene un identificador (*pid* o *rank*)
  - MPI-2 gestión dinámica

MPI: introducción

- MPI agrupa los procesos implicados en la ejecución paralela en *comunicadores*
- Un *comunicador* agrupa a procesos que pueden intercambiarse mensajes
- El *comunicador* MPI_COMM_WORLD está creado por defecto y engloba a todos los procesos

Índice

- Introducción
- MPI: conceptos generales
- **Funciones de MPI básicas**
- Comunicaciones colectivas
- Tipos de datos derivados
- Comunicadores
- Topologías
- Otros modos de comunicación
- Depurado de programas
- Introducción a MPI-2

Funciones de MPI básicas

- MPI es una librería de rutinas de comunicación
- Ficheros de cabecera
  - En C
    - #include "mpi.h" (a veces, <mpi.h>)
  - En Fortran
    - include ´mpif.h´
- Formato de las funciones de MPI
  - En C
    - int error;
    - error = MPI_Xxxxx(parámetros);
  - En Fortran
    - CALL MPI_XXXXX(parámetros, IERROR)

Funciones de MPI básicas

- Las funciones tienen parámetros
  - IN: se lee pero no se modifica
  - OUT: de salida
  - IN/OUT: se lee y se modifica
- MPI controla sus propias estructuras de datos internas
- MPI utiliza "handles" para permitir a los programadores referirse a esas estructuras
- En C se definen como ´typedef´
- En Fortran son arrays de ´INTEGER´

Funciones de MPI básicas

- Inicializar y finalizar en MPI
  - MPI_Init(int *argc, char **argv[]);
    - define (para todos los procesos) un comunicador que incluye a todos los procesos: MPI_COMM_WORLD
  - MPI_Finalize( );
  - La primera y la última función MPI que deben ejecutarse en el programa

Funciones de MPI básicas

- Identificación de procesos
  - Los procesos son ordenados y numerados consecutivamente desde 0
  - MPI_Comm_rank(MPI_Comm comm, int *myid)
    - Devuelve en `myid` el identificador del proceso dentro del comunicador `comm`
    - MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  - MPI_Comm_size(MPI_Comm comm, int *npes)
    - Devuelve en `npes` el número de procesos del comunicador `comm`
    - MPI_Comm_size(MPI_COMM_WORLD, &npes);

Funciones de MPI básicas

- Ejemplo sencillo

```
#include "mpi.h"
main(int argc, char *argv[])
{
  int myid, npes;

  MPI_Init(argc, argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  printf("Soy el nodo %d de %d\n", myid, npes);
  MPI_Finalize();
}
```

Funciones de MPI básicas

- La comunicación entre dos procesos, A y B, se realiza mediante un par de funciones:
  - Proceso A debe ejecutar una función tipo "send"
  - Proceso B debe ejecutar una función tipo "receive"
  - Si una de las funciones no se ejecuta, la comunicación no se produce (posibilidad de *deadlock*)
- Para enviar o recibir un mensaje es necesario especificar
  - A quién se envía (o de quién se recibe)
  - Datos a enviar (comienzo y longitud)
  - El tipo de datos
  - Un identificador del mensaje (tag)

Funciones de MPI básicas

- Implementaciones diferentes en función de la sincronización y buffering
  - Estándar (dependiente de la implementación)
  - Síncrona o asíncrona
  - Bloqueante o no bloqueante
  - Buffered o no buffered
  - Ready (permite acceder a protocolos rápidos)
  - Persistente

USC Computación paralela: MPI i-math CESGA

MPI básico: send y receive

- Función básica de comunicación: envío
  - int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_COMM comm)
  - Mensaje a enviar: <buf, count, datatype>
    - Tipos estándar: MPI_CHAR, MPI_FLOAT, …
  - Receptor: <dest, comm>
  - Etiqueta: <tag>, permite asociar tipos a los mensajes (clases, orden, …)
  - Implementación dependiente del sistema

Computación paralela: MPI

MPI básico: send y receive

- Función básica de comunicación: recepción
  - int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int fuente, int tag, MPI_COMM comm, MPI_Status status)
  - Mensaje a recibir: <buf, count, datatype>
    - Tipos estándar: MPI_CHAR, MPI_FLOAT, …
  - Receptor: <fuente, comm>
  - Etiqueta: <tag>
  - Estado: <status>, información de control sobre el mensaje recibido
  - MPI_Revc es bloqueante

Computación paralela: MPI

MPI básico: send y receive

- Consideraciones
  - El tamaño del mensaje en MPI_Recv() debe ser al menos igual que el de MPI_Send()
    - Se puede obtener con
      MPI_Get_count(status, datatype, &count)
  - En MPI_Recv()
    - El origen puede ser MPI_ANY_SOURCE
    - La etiqueta puede ser MPI_ANY_TAG
  - status es una estructura con información del mesaje
    - status.MPI_SOURCE: indica el emisor del mensaje
    - status.MPI_TAG: indica la etiqueta del mensaje recibido
    - status.MPI_ERROR: código de error

Funciones de MPI básicas

- Ejemplo sencillo: el proceso 0 envía **mensaje** al resto

```
#include "mpi.h"
main(int argc, char *argv[])
{
  int myid, npes, i;

  MPI_Init(argc, argv);
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if(myid = = 0)
    for(i=1; i<npes;i++)
       MPI_Send(mensaje, tamaño, MPI_CHAR, i, tipo, MPI_COMM_WORLD);
    else
       MPI_Recv(mensaje, tamaño, MPI_CHAR, 0, tipo, MPI_COMM_WORLD, &estado);
  printf("Soy el nodo %d: %s\n", myid, mensaje);

  MPI_Finalize();
}
```

MPI básico: entrada/salida

- Entrada de datos y salida
  - Normalmente, sólo un proceso tiene acceso al teclado y pantalla
    - Ese proceso leerá los datos y los distribuye
    - Al finalizar, este proceso recopilará los datos
  - Ejemplo: leer datos y distribuir

```
If (myid == 0){
    lee_datos();
    distribuye_datos();
}else
    recibe_datos();
```


MPI básico: tipos de datos

- Se definen los siguientes tipos de datos MPI:

MPI_CHAR	MPI_UNSIGNED_LONG
MPI_SHORT	MPI_FLOAT
MPI_INT	MPI_DOUBLE
MPI_LONG	MPI_LONG_DOUBLE
MPI_UNSIGNED_CHAR	MPI_BYTE
MPI_UNSIGNED_SHORT	MPI_PACKED
MPI_UNSIGNED	

- Corresponden a los de C, pero se añaden el tipo byte, y el empaquetado, que permite enviar simultáneamente datos de distintos tipos.

MPI básico: tiempo de ejecución

- Funciones para medir tiempos de ejecución
  - double MPI_Wtime()
    - Tiempo transcurrido desde un instante anterior en segundos
  - double MPI_Wtick()
    - Devuelve la resolución de MPI_Wtime() en segundos
  - Para medir el tiempo de ejecución:

```
T1 = MPI_Wtime();
…
…
T2 = MPI_Wtime();
Tiempo = T2 – T1;
```


Índice

- Introducción
- MPI: conceptos generales
- Funciones de MPI básicas
- **Comunicaciones colectivas**
- Tipos de datos derivados
- Comunicadores
- Topologías
- Otros modos de comunicación
- Depurado de programas
- Introducción a MPI-2

Comunicaciones colectivas

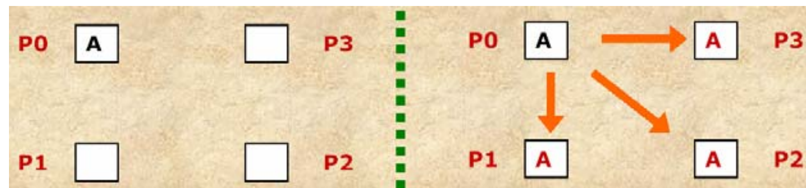
- Muchas aplicaciones requieren operaciones de comunicación en las que intervienen muchos procesos
- Comunicación colectiva (o global): participan todos los procesos de un comunicador
  - Se podrían hacer con comunicaciones punto a punto
- Ejemplo: una radiación (broadcast), envío de datos de un proceso a todos los demás
  - Se podría hacer en un bucle

Comunicaciones colectivas

- Funciones de comunicación colectivas:
  - Son bloqueantes
  - Todos los procesos del comunicador deben ejecutar la función
  - El buffer de recepción debe ser del tamaño exacto
- Tipos:
  - Movimiento de datos
  - Cálculo en grupo
  - Sincronización

Comunicaciones colectivas

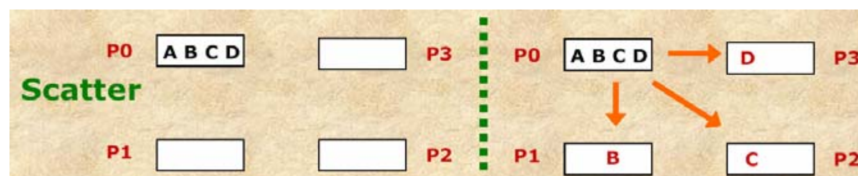
- Movimiento de datos: radiación
  - Envío de datos desde un proceso raíz a todos los demás



  - Se podría realizar con un lazo y comunicaciones punto a punto
  - int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

Comunicaciones colectivas

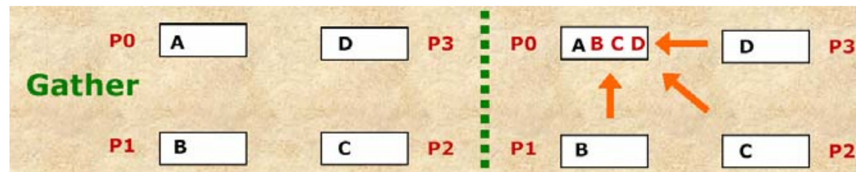
- Movimiento de datos: scatter
  - Scatter: distribución de datos de un proceso entre todos los demás



  - MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Comunicaciones colectivas

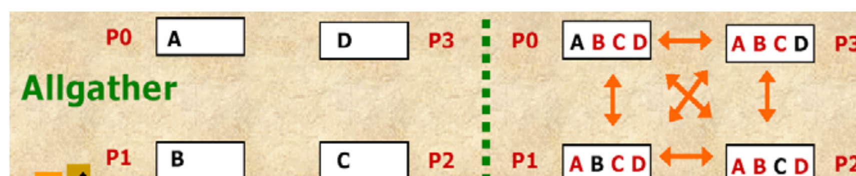
- Movimiento de datos: gather
  - Gather: recolección de datos de todos los procesos en uno de ellos



  - MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Comunicaciones colectivas

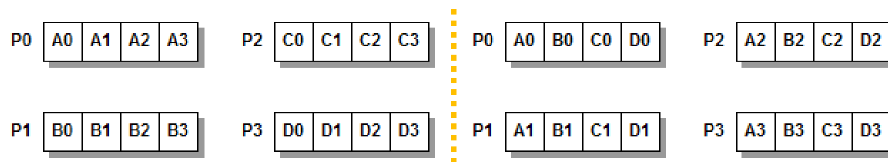
- Movimiento de datos: allgather
  - Allgather: gather de todos a todos
  - Al final todos los procesos disponen de todos los datos recolectados



  - MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

Comunicaciones colectivas

- Movimiento de datos: MPI_Alltoall
  - Envío general de todos a todos
  - int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm);

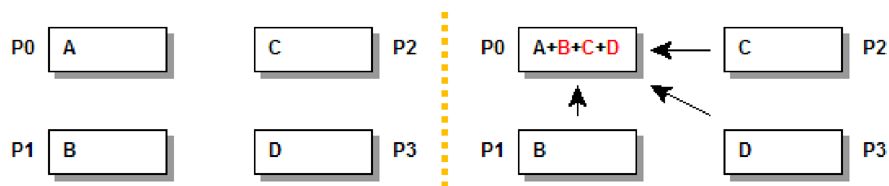


Comunicaciones colectivas

- Movimiento de datos: variantes "v"
  - MPI_Scatterv, MPI_Gatherv, MPI_Allgatherv, MPI_Alltoallv
  - Envían un número de datos diferente para cada proceso
    - En las otras aparece "count", que es un entero
    - Variantes "v", el equivalente son dos vectores
    - Ejemplo:
      - int MPI_Alltoallv(void *sendbuf, **int* sendcounts**, **int* sdispls**, MPI_Datatype sendtype, void *recvbuf, **int* recvcounts**, **int* rdispls**, MPI_Datatype recvtype, MPI_Comm comm);

Comunicaciones colectivas

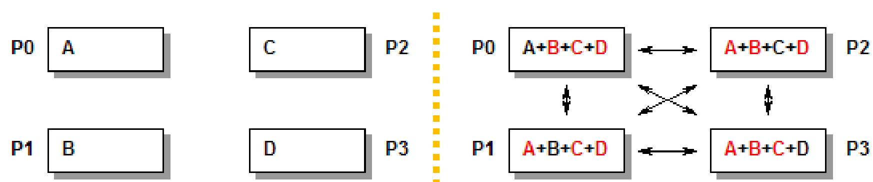
- Cálculo en grupo: reducción
  - Reduce: una operación de cálculo con los datos de cada procesador, dejando el resultado en uno (raíz)
  - MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  - Depende de la implementación (árbol, etc.)



Comunicaciones colectivas

- Operaciones de reducción:

  MPI_MAX MPI_MIN MPI_SUM MPI_PROD
  MPI_LAND MPI_BAND MPI_LOR MPI_BOR
  MPI_LXOR MPI_BXOR MPI_MAXLOC MPI_MINLOC
  Operaciones definidas por el usuario
- Variante MPI_Allreduce
  - Deja el resultado en todos los procesos



Computación paralela: MPI

Comunicaciones colectivas

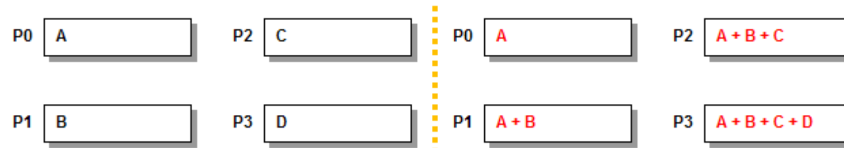
- Operaciones de reducción definidas por el usuario
  - La función debe seguir el esquema
    - int User_function(void *invec, void *inoutvec, int *len, MPI_Datatype datatype)
  - La función se registra con
    - int MPI_Op_create(MPI_User_function *function, int conmute, MPI_Op *op)
  - La operación se elimina con
    - int MPI_Op_free(MPI_Op *op)

Comunicaciones colectivas

- Sincronización: barreras
  - Sincronización global entre los procesos del comunicador
  - Se detiene la ejecución de los procesos hasta que todos los procesos llegan a ese punto
  - Útil para hacer medidas de tiempo de ejecución de partes concretas de código
  - int MPI_Barrier(MPI_Comm comm)

Comunicaciones colectivas

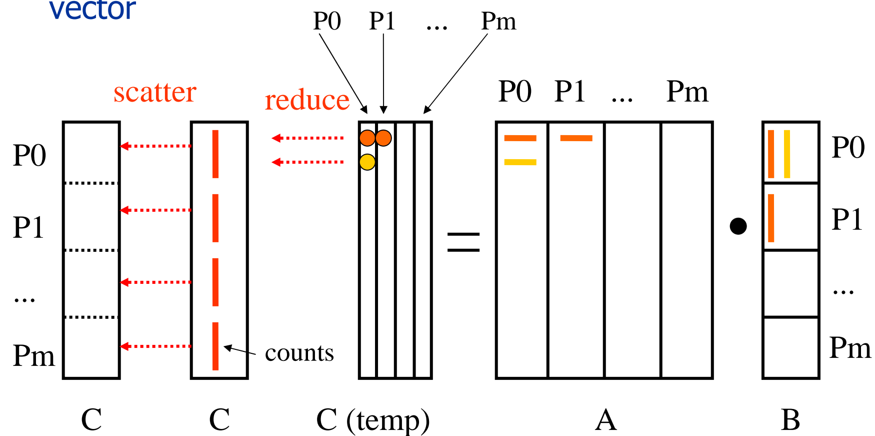
- Otras operaciones
  - Scan (Reducción parcial)
    - MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)



  - Reducción y scatter
    - Combina MPI_Reduce y MPI_Scatterv
    - MPI_Reduce_scatter (void *sendbuf, void *recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Comunicaciones colectivas

- MPI_Reduce_scatter en multiplicación de matriz por vector



Comunicaciones colectivas

- Ejemplo: v(i) = v(i) * Σv(j)

```
sum = 0;
for(j=0;j<N;j++) sum = sum + v[j];
for(i=0;i<N;i++) v[i] = v[i] * sum;
```

1.- Leer N (el proceso 0)
2.- Broadcast de N o de N/NPES (tamaño del vector local)
3.- Scatter del vector V (trozo correspondiente a cada proceso)
4.- Cálculo local de la suma parcial (cada proceso calcula la suma de su trozo)
5.- Allreduce de las sumas parciales (al final todos tienen el total)
6.- Cálculo local de v(i)*sum
7.- Gather del vector local (al proceso 0)
8.- Salida del resultado (el proceso 0)

Índice

- Introducción
- MPI: conceptos generales
- Funciones de MPI básicas
- Comunicaciones colectivas
- **Tipos de datos derivados**
- Comunicadores
- Topologías
- Otros modos de comunicación
- Depurado de programas
- Introducción a MPI-2

Tipos de datos derivados

- Comunicación entre procesos depende de:
  - Latencia
  - Ancho de banda
  - Modelo sencillo: T = latencia + BW*tamaño
- Preferible enviar menos mensajes y más grandes
- Comunicaciones punto a punto
  - Indican la dirección de comienzo
  - Número de elementos
  - Restricción: contiguos y del mismo tipo

Datos derivados

- Ejemplo: enviar la fila 2 de una matriz de P0 a P1

```
float A[10][10];

if (pid==0)
      MPI_Send(&A[2][0], 10, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
else if (pid==1)
      MPI_Recv(&A[2][0], 10, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
&info);
```

- ¿Cómo enviar una columna?
- ¿Cómo enviar datos de tipos diferentes?

  ⇒ Definiendo tipos de datos derivados

Datos derivados

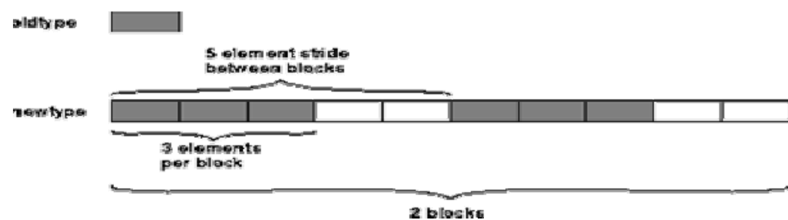
- Datos derivados en MPI
  - Con datos homogéneos
    - Grupos de datos contiguos
    - Grupos de datos con un *stride* (espaciado) constante
    - Grupos de datos con un *stride* (espaciado) variable
  - Con datos heterogéneos
    - Estructuras
- Aunque para enviar datos heterogéneos
  - Se pueden empaquetar (MPI_Pack(), MPI_Unpack())

Datos derivados

- Con datos homogéneos:
  - Grupos contiguos
    - Se crea un tipo de datos de varios datos contiguos
    - int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
  - Grupos de datos con un *stride* constante
    - Se crea un tipo de datos que consiste en:
      - Varios bloques de datos
      - Un espaciado constante entre bloques
    - int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

Datos derivados

- Ejemplo:
  - MPI_Type_vector(count, blocklength, stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
  - count = 2
  - blocklenght = 3
  - stride = 2



Datos derivados

- Ejemplo: enviar la columna 2 de una matriz de P0 a P1

```
float A[10][10];
MPI_Datatype T_columna;

MPI_Type_vector(10, 1, 10, MPI_FLOAT, &T_columna);
MPI_Type_commit(&T_columna);
if (pid==0)
      MPI_Send(&A[0][2], 1, T_columna, 1, 0, MPI_COMM_WORLD);
else if (pid==1)
      MPI_Recv(&A[0][2], 1, T_columna, 0, 0, MPI_COMM_WORLD,
&info);
MPI_Type_free (&T_columna);
```

Datos derivados

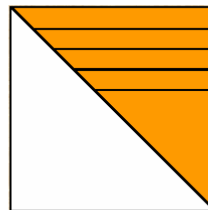
- Con datos homogéneos:
  - Grupos de datos con un *stride* variable
    - Se crea un tipo de datos "indexed"
    - int MPI_Type_indexed(int count, int *blocklength, int *stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
    - count es un entero que indica el número de bloques
    - blocklength es un array que contiene el número de componentes de cada "elemento" que forma el nuevo tipo
    - stride es un array que contiene el desplazamiento necesario para acceder desde el comienzo a cada "elemento"

Datos derivados

- Ejemplo: enviar el triángulo superior de una matriz de P0 a P1

```
float A[N][N], T[N][N];
MPI_Datatype T_tri;

for(i=0; i<N; i++) {
      long_bl[i] = N - i;
      desp[i] = (N+1) * i;
}
MPI_Type_indexed (N, long_bl, desp, MPI_FLOAT, &T_tri);
MPI_Type_commit (&T_tri);
if (pid==0) MPI_Send(A, 1, T_tri, 1, 0, MPI_COMM_WORLD);
else if (pid==1) MPI_Recv(T, 1, T_tri, 0, 0, MPI_COMM_WORLD,
&info);
```



Datos derivados

- Con datos heterogéneos:
  - Utilidad: lectura de parámetros (de tipos diferentes) del teclado y después radiarlos a todos los procesos
    - Se agrupan en un único tipo y se realiza un único envío
  - Estructuras
  - int MPI_Type_struct(int count, int *blocklength, MPI_Aint *displ, MPI_Datatype *array_oldtypes, MPI_Datatype *newtypes)
    - `count` indica el número de tipos
    - `blocklength` es una array que indica el número de datos de cada tipo
    - `displ` es un array con las direcciones de comienzo de cada tipo
    - `array_oldtypes`, array de MPI_Datatype que indica los distintos tipos

Datos derivados

- Ejemplo: dos float (a, b) y un int (c)

```
int longi[3];
MPI_Aint despl[3], dir1, dir2;
MPI_Dataype tipos[3], T_struct;

longi[0] = longi[1] = longi[2] = 1;
tipos[0] = tipos[1] = MPI_FLOAT; tipos[2] = MPI_INT;
despl[0] = 0;
MPI_Address (a, &dir1); MPI_Address (b, &dir2);
despl[1] = dir2 - dir1;
MPI_Address (c, &dir2);
despl[2] = dir2 - dir1;
MPI_Type_struct(3, longi, despl, tipos, &T_struct);
MPI_Type_commit(&T_struct);
```

Datos derivados

- Empaquetamiento de datos
  - Almacenarlos en posiciones de memoria consecutivas.
  - Dos funciones efectúan el empaquetado en el emisor y el desempaquetado en el receptor
    - int MPI_Pack (void *inbuf, int incount, MPI_Datatype datatype, void *outbuf, int outsize, int *position, MPI_Comm comm);
      - Añade incrementalmente datos a `outbuf`
      - `position` es un argumento de entrada y salida: se va actualizando a medida que se añaden datos
    - int MPI_Unpack (void *inbuf, int insize, int *position, void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm);

Datos derivados

- Ejemplo: dos float (a, b) y un int (c)

```
char buffer[TAM_MAX];

if (pid==0) {
     posicion = 0;
     MPI_Pack(a, 1, MPI_FLOAT, buffer,TAMAÑO, &posicion, MPI_COMM_WORLD);
     MPI_Pack(b, 1, MPI_FLOAT, buffer, TAMAÑO, &posicion, MPI_COMM_WORLD);
     MPI_Pack(c, 1, MPI_INT, buffer, TAMAÑO, &posicion, MPI_COMM_WORLD);
     }
MPI_Bcast(buffer, TAMAÑO, MPI_PACKED, 0, MPI_COMM_WORLD);
If(pid!=0) {
     posicion = 0;
     MPI_Unpack(buffer, TAMAÑO, &posicion, a, 1, MPI_FLOAT, MPI_COMM_WORLD);
     MPI_Unpack(buffer, TAMAÑO, &posicion, b, 1, MPI_FLOAT, MPI_COMM_WORLD);
     MPI_Unpack(buffer, TAMAÑO, &posicion, c, 1, MPI_INT, MPI_COMM_ WORLD);
}
```

Datos derivados

- Resumen
  - Un tipo derivado es una estructura que se crea en ejecución y que se puede usar en las comunicaciones
  - MPI_Type_contiguous, vector, indexed, struct
  - Después de crearlos hay que "darlos de alta" antes de utilizarlos con la función MPI_Type_commit
- Usos
  - En general, datos consecutivos en un array
  - Datos derivados, si no son consecutivos o son heterogéneos, y se envían muchas veces
  - Si son datos heterogéneos y se envían pocas veces, se pueden empaquetar

Índice

- Introducción
- MPI: conceptos generales
- Funciones de MPI básicas
- Comunicaciones colectivas
- Tipos de datos derivados
- **Comunicadores**
- Topologías
- Otros modos de comunicación
- Depurado de programas
- Introducción a MPI-2

Comunicadores

- Un comunicador agrupa un conjunto de procesos entre los que pueden intercambiarse información.
- El comunicador MPI_COMM_WORLD está predefinido y engloba a todos los procesos.
- Pueden definirse nuevos comunicadores con subgrupos de procesos

Comunicadores

- Un comunicador está formado, al menos, por un grupo de procesos y un contexto
  - El contexto define un espacio propio e identificado de comunicación
  - Un proceso puede formar parte de diferentes comunicadores, y tendrá un identificador para cada contexto del que forme parte
  - Un comunicador puede incluir más datos asociados, tales como una topología, …

Comunicadores

- Creación de un comunicador: tres pasos
  - Grupo asociado a un comunicador
    - int MPI_Comm_group(MPI_Comm comm, MPI_Group *grupo)
  - Crear grupos
    - int MPI_Group_incl(MPI_Group grupo, int newsize, int *ids, MPI_Group *newgrupo)
  - Crear comunicador asociado a un grupo
    - MPI_Comm_create(MPI_Comm comm, MPI_Group newgrupo, MPI_Comm *newcomm)

Comunicadores

- Ejemplo: fila 0 de una malla de f x c procesos

```
MPI_Comm C_fila0;
MPI_Group grupo, grupo_fila0;
// lista de los procesos del nuevo comunicador
for(proc=0; proc<c; proc++) pids[proc] = proc;

MPI_Comm_group(MPI_COMM_WORLD, &grupo);
MPI_Group_incl(grupo, c, pids, &grupo_fila0);
MPI_Comm_create(MPI_COMM_WORLD, grupo_fila0, &C_fila0);
...
MPI_Comm_rank(C_ fila0, &pid_f0);
//El 0 de la fila 0, radia a los demás de la fila 0
MPI_Broadcast(&A[0][0], tam, MPI_FLOAT, 0, C_ fila0);
```

Comunicadores

- Se pueden crear varios comunicadores simultáneamente
  - int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
    - Función colectiva
    - Todos los procesos con el mismo "color" van al mismo comunicador
    - Los procesos se ordenan dentro de cada comunicador por la "clave"

Comunicadores

- Ejemplo: comunicadores para cada fila de una malla de f x c procesos

```
// pid = identificador en el comunicador global

mi_fila = pid/c;
MPI_Comm_split(MPI_COMM_WORLD, mi_fila, pid, &C_mi_fila);
```

  - Todos ejecutan la función
  - Si un proceso no va a formar parte de los nuevos comunicadores, llamarán a la función con "color" MPI_UNDEFINED

Índice

- Introducción
- MPI: conceptos generales
- Funciones de MPI básicas
- Comunicaciones colectivas
- Tipos de datos derivados
- Comunicadores
- **Topologías**
- Otros modos de comunicación
- Depurado de programas
- Introducción a MPI-2

Topologías

- Un comunicador puede incluir, además del grupo y el contexto, otro tipo de información; por ejemplo, una topología (virtual)
- No tiene relación con la topología física (red de comunicación)
- Mecanismo que permite asociar distintos esquemas de direccionamiento dentro de un grupo de procesos
- Dos tipos: cartesiana y basada en grafos

Topologías

- Ventajas
  - Permiten numerar convenientemente los procesos.
  - El esquema de numeración permite mejorar los patrones de comunicación.
  - Simplifica la escritura del código.
  - Permite a MPI optimizar la comunicación.
- Uso
  - La creación de una topología produce un nuevo comunicador
  - MPI tiene funciones de mapeo que generan las nuevas coordenadas del proceso en la topología

Topologías

- Tipos de topologías
  - Cartesiana
    - Cada procesador está conectado a sus vecinos
    - Sus fronteras pueden ser cíclicas o no
    - Los procesos se identifican por sus coordenadas cartesianas
  - Topología basada en grafos
- Creación de una topología cartesiana
  - int MPI_Cart_create(MPI_COMM comm, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)
  - `periods` indica si la dimensión es cíclica o no
  - `reorder` indica a MPI que puede reordenar los procesos para optimizar las comunicaciones

Topologías

- Por ejemplo, para algoritmos 2D, podemos hacer abstracción de la estructura de procesos y definir una malla (por ejemplo, de 4x4 si tenemos 16 procesadores)
  - Número de dimensiones, 2
  - Elementos por dimensión, 4
  - ¿Malla o toro?
  - Optimización (ajuste a la red física)

Topologías

- Ejemplo: añadir un toro al comunicador global

```
n_dim = 2; opt = 1;
dims[0] = dims[1] = 4;
toro[0] = toro[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, n_dim, dims, toro, opt,
&C_toro);
```

  - Los procesos del nuevo comunicador llevan un nuevo identificador y tienen asociadas coordenadas en la topología definida

Topologías

- Funciones de obtención de coordenadas
  - Obtención el identificador a partir de las coordenadas de la malla
    - int MPI_Cart_rank(MPI_Comm comm, int *coordenadas, int *myid)
  - Obtención de las coordenadas a partir del identificador
    - int MPI_Cart_coords(MPI_Comm comm, int myid, int maxdims, int *coordenadas)

Topologías

- Particionamiento cartesiano
  - int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
    - Divide la malla en subtopologías
    - Se crea un nuevo comunicador en cada región y, por tanto, cada nueva región puede realizar sus propias comunicaciones colectivas
    - Alternativa a MPI_Comm_split
  - Ejemplo: partición por filas

```
MPI_comm  *fila_comm;  int var_coords[2];
var_coords[0] = 0  ( 0 porque la coordenada "x" no varía en cada fila)
var_coords[1] = 1  ( 1 porque la coordenada "y" varía en cada fila)
MPI_Cart_sub( grid_comm , var_coords , fila_comm)
```

Topologías

- Otras funciones de topologías cartesianas
  - Obtención de coordenadas y características del comunicador cartesiano
    - int MPI_Cart_get(MPI_Comm grid_comm , int maxdims, int *dims , int *periods, int *coords)
  - Obtención de procesos "vecinos"
    - int MPI_Cart_shift(MPI_Comm grid_comm , int direction, int displ , int *id_source, int *id_dest)
    - Obtiene el identificador del proceso de origen y el de destino (para usar en un send/recv) entre procesos separados por `displ` procesos en la dirección `direction`.

Índice

- Introducción
- MPI: conceptos generales
- Funciones de MPI básicas
- Comunicaciones colectivas
- Tipos de datos derivados
- Comunicadores
- Topologías
- **Otros modos de comunicación**
- Depurado de programas
- Introducción a MPI-2

Otros modos de comunicación

- Los modos estándar de envío y recepción son bloqueantes
  - En implementaciones puede perder eficiencia
  - En aplicaciones puede producir bloqueos (*deadlock*)
  - Por ejemplo:

P0	P1
send_to_1	send_to_0
recv_from_1	recv_from_0

    - Obviamente, habría que cambiar el orden

Otros modos de comunicación

- Para comunicaciones en "anillo"
  - int MPI_Sendrecv(void *sendbuff, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuff, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPIComm comm, MPI_Status *status)
  - Parámetros del MPI_send y MPI_recv.
  - Efectúa una operación de envío y recepción en el orden adecuado para evitar el *deadlock*

Otros modos de comunicación

- Ejemplo: comunicación en anillo

```
src = myid-1;
dest = myid+1;
if (src < 0) src += num_nodes;
if (dest >= num_nodes) dest -= num_nodes;

MPI_Sendrecv(msg_out, msg_len, MPI_INT, dest, tag, msg_in,
   msg_len, MPI_INT, src, tag, MPI_COMM_WORLD, &status);
```


Otros modos de comunicación

- Modelos de comunicación
  - Bloqueante
  - No bloqueante
- Modos de comunicación
  - Básico
  - Con buffer (*buffered*)
  - Síncrono
  - Preparado (*ready*)
  - Persistente

Otros modos de comunicación

- Comunicaciones no bloqueantes ("inmediatas")
  - Copian el mensaje en buffer y retornan inmediatamente, sin esperar a que se efectúe la comunicación
  - Funciones básicas no bloqueantes
    - int MPI_Isend(…, MPI_Request *request)
    - int MPI_Irecv(…, MPI_Request *request)
    - La estructura (*handle*) MPI_Request es una especie de recibo de la operación solicitada
      - Conocer si la operación ha terminado o no

Computación paralela: MPI

Otros modos de comunicación

- Saber si se completa o no la comunicación
  - Se utiliza el *handle* MPI_request
  - int MPI_Wait(MPI_request *request, MPI_Status *status)
    - Espera a que la comunicación se haya efectuado
    - MPI_Isend + MPI_Wait = MPI_Send, pero permite realizar cálculos entre el envío y la espera
  - int MPI_Test(MPI_request *request, int *flag, MPI_Status *status)
    - En `flag` devuelve un 0 si la comunicación no se ha completado

Computación paralela: MPI

Otros modos de comunicación

- Ejemplo: comunicación no bloqueante

```
int flag = 0;
…
MPI_Isend(buffer, muchos, MPI_INT, dest, tag, MPI_COMM_WORLD,
   &req);
while(!flag && hay_tareas) {
   /* realizar cálculos */
   MPI_Test(&req, &flag, &status);
}
MPI_Wait(&req, & status);
```


Otros modos de comunicación

- Saber si se completan múltiples comunicaciones
  - Funciones de espera
    - MPI_Waitall, espera a todas
    - MPI_Waitany, espera a alguna
    - MPI_Waitsome, espera a un conjunto determinado
  - Funciones de comprobación
    - MPI_Testall, comprueba todas
    - MPI_Testany, comprueba alguna
    - MPI_Testsome, comprueba un conjunto determinado
- Funciones de chequeo (sin un MPI_Request)
    - MPI_Probe, MPI_Iprobe, comprueban si llegan mensajes

Otros modos de comunicación

- Envío con buffer (*buffered*)
  - El mensaje se copia en un buffer
    - Evita el bloqueo del proceso
    - El buffer de usuario se puede utilizar mientras se prepara el destinatario
  - MPI_Bsend, con los mismos argumentos que MPI_Send
  - El usuario debe asignar un buffer de salida
    - int MPI_Buffer_attach(void *buffer, int size)
      - Indica al sistema que lo emplee como buffer
    - int MPI_Buffer_detach(void *buffer, int *size)
      - El buffer se recupera para otros usos

Otros modos de comunicación

- Envío preparado (*ready*)
  - Se usa si el receptor está preparado para una recepción inmediata
    - No hay copias adicionales
    - Hay implementaciones que lo tratan como un envío normal, pero permiten optimizar el protocolo, si es posible
    - Si el receptor no está preparado, el comportamiento es impredecible
  - MPI_Rsend, con los mismos argumentos que MPI_Send

Otros modos de comunicación

- Envío síncrono
  - La función de envío no retorna hasta que el receptor confirma la recepción y comienza a leer el mensaje
  - En principio, no necesita buffer del sistema
  - MPI_Ssend, con los mismos argumentos que MPI_Send
- Todos los modos tienen versión no bloqueante
  - Básico: MPI_Isend, MPI_Irecv
  - MPI_Ibsend, MPI_Irsend, MPI_Issend

Otros modos de comunicación

- Comunicación persistente
  - Mejora el rendimiento cuando se envía repetidamente mensajes con los mismos argumentos
  - Dos pasos:
    - Crear el contexto
      - int MPI_Send_init(..., MPI_Request *request)
        - Modo de comunicación básico
      - int MPI_Recv_init(..., MPI_Request *request)
    - Enviar el mensaje
      - int MPI_Start(MPI_Request *request);
    - MPI_Send = MPI_Send_init + MPI_Start
    - Para crear el contexto para comunicaciones de otros modos
      - MPI_Bsend_init, MPI_Isend_init, MPI_Ssend_init

Otros modos de comunicación

- Resumen
  - La comunicación es determinante en el rendimiento de un sistema paralelo de memoria distribuida
  - Por ello, MPI ofrece muchas alternativas
  - Uso recomendado:
    - MPI_Send: la alternativa más habitual
    - MPI_Isend: si se necesitan comunicaciones no bloqueantes
    - MPI_Ssend: cuando es posible, ofrece los mejores resultados porque no se utilizan buffers intermedios
    - El resto (MPI_Bsend y MPI_Rsend) para casos especiales

Índice

- Introducción
- MPI: conceptos generales
- Funciones de MPI básicas
- Comunicaciones colectivas
- Tipos de datos derivados
- Comunicadores
- Topologías
- Otros modos de comunicación
- **Depurado de programas**
- Introducción a MPI-2

Depurado de programas

- Depurado de programas paralelos
  - Más difícil
  - Los programas paralelos pueden ofrecer resultados no deterministas
  - Difícil de predecir el comportamiento de un programa erróneo
    - Puede cambiar de sistema a sistema
  - Muchos de los errores no tienen nada que ver con el paralelismo

Depurado de programas

- Depurado de programas paralelos
  - Algunos errores típicos
    - Intentar recibir datos si haberlos enviado

      ⇒ *deadlock*
    - Parámetros incorrectos en funciones de comunicación
  - Conveniente imprimir los parámetros antes de enviarlos y cuando se reciben
  - La introducción de trazas puede modificar el comportamiento del programa

Depurado de programas

- MPI proporciona rutinas y estructuras para manejar errores (*error handler*)
  - MPI_ERRORS_ARE_FATAL
  - MPI_ERRORS_RETURN
  - Están asociados a los comunicadores
    - Los comunicadores llevan parámetros asociados, por ejemplo, las topologías
  - int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler);
    - Asocia un *error handler* al comunicador
  - int MPI_Error_string(int errorcode, char *string, int *resultlen);
    - Devuelve el string asociado al código de error

Depurado de programas

- Ejemplo:

```
char mens_err[MPI_MAX_ERROR_STRING];
cod_err = MPI_Errhandler_set(MPI_COMM_WORLD,
   MPI_ERRORS_RETURN);

cod_err = MPI_Broadcast(&x, 1, MPI_INT, 0, comm);
if (cod_err != MPI_SUCCESS) {
   MPI_Error_string(cod_err, mens_err, &longitud_mens);
   fprintf(stderr, "Error MPI_Broadcast = %s\n", mens_err);
   MPI_Abort(MPI_COMM_WORLD, -1);      // -1: error_code
}
```

Índice

- Introducción
- MPI: conceptos generales
- Funciones de MPI básicas
- Comunicaciones colectivas
- Tipos de datos derivados
- Comunicadores
- Topologías
- Otros modos de comunicación
- Depurado de programas
- **Introducción a MPI-2**

MPI-2

- Nuevas características
  - Mejoras y extensiones a MPI-1
  - Gestión dinámica de procesos
  - Acceso remoto a memoria
  - Entrada/salida paralela
  - Interacción con *threads*
  - Interoperatividad entre lenguajes
  - …

MPI-2: extensiones a MPI-1

- Extensiones a MPI-1
  - Nuevas funciones de tipos de datos derivados
    - Grupos de datos con un *stride* variable, como MPI_Type_create_indexed, pero con *blocklength* constante
      - MPI_Type_create_indexed_block
      - Utilizado en códigos con indirecciones
    - Constructor de subarrays
      - MPI_Type_create_subarray
    - Constructor de array distribuido
      - MPI_Type_create_darray
      - Para hacer distribuciones tipo HPF
    - Funciones para decodificar tipos de datos derivados
      - MPI_Type_get_envelope, MPI_Type_get_contents

MPI-2: extensiones a MPI-1

- Extensiones a MPI-1
  - Nuevos tipos de datos predefinidos
    - MPI_WCHAR, MPI_SIGNED_CHAR, MPI_UNSIGNED_LONG_LONG
  - Reserva dinámica de memoria
    - Mecanismos especiales para reservar memoria
    - En muchos sistemas, se pueden optimizar las operaciones de pase de mensajes y acceso remoto a memoria
    - MPI_Alloc_mem, MPI_Free_mem
    - Funciones para permitir o no el acceso remoto a memoria
      - MPI_Win_lock, MPI_Win_unlock

MPI-2: extensiones a MPI-1

- Extensiones a MPI-1
  - Operaciones colectivas extendidas
    - En MPI-1 las operaciones colectivas funcionan con comunicadores (intracomunicadores en MPI-2)
    - Extendidas a intercomunicadores
    - Operaciones nuevas
      - MPI_Alltoallw, generalización de MPI_Alltoallv
      - MPI_Exscan

MPI-2: gestión de procesos

- Modelo de procesos en MPI-2
  - Permite la creación y terminación cooperativa de procesos una vez iniciada la aplicación MPI
  - Proporciona un mecanismo para comunicar procesos recién creados con los existentes
  - Proporciona mecanismos para comunicar dos aplicaciones MPI existentes, aunque una de ellas no haya iniciado a la otra
  - Útil en muchas aplicaciones
    - Aplicaciones secuenciales con módulos paralelos
    - Que requieren fijar el número y tipo de procesos en tiempo de ejecución

MPI-2: gestión de procesos

- Iniciar procesos
  - Inician procesos MPI y establecen comunicación con ellos devolviendo un intercomunicador (MPI_Comm)
  - int MPI_Comm_spawn(char *command, char *argv[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int *err_codes)
  - Para conocer el intercomunicador padre
    - int MPI_Comm_get_parent(MPI_Comm *parent)
  - int MPI_Comm_spawn_multiple
    - Inicia procesos pero con comandos diferentes
    - Los argumentos son arrays (de comandos, argumentos,…)

MPI-2: RMA

- Acceso remoto a memoria (RMA, *Remote Memory Access*)
  - Extiende los mecanismos de comunicación
  - Permite que un proceso especifique todos los parámetros de la comunicación, unilateralmente
    - O bien desde el emisor
    - O bien desde el receptor
  - Send/Recv necesitan operaciones "conjuntas", coincidencia de parámetros, etc.
  - Funciones básicas: MPI_Put (escritura remota), MPI_Get (lectura remota), MPI_Accumulate (actualización remota)

MPI-2: RMA

- Acceso remoto a memoria
  - El usuario debe imponer el orden correcto de los accesos, introduciendo sincronizaciones
  - Permite optimizar las comunicaciones usando mecanismos proporcionados por sistemas de memoria compartida
  - Proceso origen: proceso que realiza la llamada
  - Proceso destino: proceso cuya memoria se accede
  - Proceso fuente: el que "envía" los datos
  - Proceso receptor: el que "recibe" los datos
    - Operación put: fuente = origen, receptor = destino
    - Operación get: fuente = destino, receptor = origen

MPI-2: RMA

- Inicialización
  - Permite a los procesos de un comunicador especificar una "ventana" en su memoria que sea accesible de forma remota
  - int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
    - El objeto `win` representa el grupo de procesos y las ventanas de memoria con sus atributos
    - La zona de memoria comienza en `base` y es de `size` bytes
    - `disp_unit` es para facilitar la aritmética de direcciones
  - int MPI_Win_free(MPI_Win *win)

MPI-2: RMA

- Tres funciones de comunicación
  - MPI_Put transfiere datos desde el que llama a la función (origen) al destino (acceso en su memoria)
  - MPI_Get transfiere datos desde el destino al que llama a la función
  - MPI_Accumulate actualiza posiciones en la memoria destino
  - Son no bloqueantes: la transferencia puede continuar después del retorno de la función
  - Finalizan con una llamada a una función de sincronización

MPI-2: RMA

- Funciones de comunicación
  - int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target, int target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win *win)
  - Equivalente a una comunicación
  - MPI_Send(origin_addr, origin_count, origin_datatype, target, tag, comm)
  - MPI_Recv(target_addr, target_count, target_datatype, source, tag, comm)
    - Con target_addr = base + target_disp X disp_unit
    - base y disp_unit son los atributos de la ventana de memoria

MPI-2: RMA

- Funciones de comunicación
  - int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target, int target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win *win)
  - Mismos argumentos que MPI_put
  - int MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target, int target_disp, int target_count, MPI_Datatype target_datatype, **MPI_Op op**, MPI_Win *win)
    - Cualquier operación de MPI_Reduce
- Funciones de sincronización
  - Necesarias para completar las comunicaciones

MPI-2: E/S paralela

- Entrada/salida paralela
  - Permitir el particionamiento de los datos de un fichero entre procesos
  - Transferencia de información entre ficheros y la memoria de los procesos
  - Además
    - E/S asíncrona
    - Accesos con espaciado
    - Mantenimiento de la consistencia por múltiples accesos
    - …
  - Importante los tipos de datos derivados

MPI-2: E/S paralela

- Manejo de ficheros
  - Abrir un fichero
    - int MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *fh)
    - Operación colectiva: todos los procesos deben proporcionar el nombre que referencie al mismo fichero y el mismo modo
  - Cerrar un fichero
    - int MPI_File_close(MPI_File *fh)
    - Operación colectiva
      - Primero sincroniza el estado del fichero (equivalente a MPI_File_sync) y después lo cierra
  - Otras funciones
    - Borrar, obtener información, cambiar de tamaño,...

MPI-2: E/S paralela

- Funciones de acceso a datos en función del:
  - Posicionamiento
    - Offset explícito
    - Puntero a fichero individual
    - Puntero a fichero compartido
  - Sincronismo
    - Bloqueante o no bloqueante
  - Coordinación
    - Colectiva o no colectiva
  - Tipo de operación
    - Lectura o escritura

MPI-2: E/S paralela

- Ejemplos:
  - int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
    - Lee datos de un fichero al comienza de la posición dada por el offset
  - int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
    - Análoga a la anterior, pero colectiva

MPI-2: interacción con threads

- Interacción con threads
  - Un proceso MPI puede ser multithreaded
  - Cada thread puede hacer llamadas a funciones MPI
    - Los identificadores identifican a los procesos, no a los threads
  - Funciones bloqueantes llamadas desde un thread, sólo bloquean el thread
  - Se necesita iniciar MPI y el entorno para gestionar threads
    - En lugar de MPI_Init se usa MPI_Init_thread

MPI-2: interacción con threads

- Interacción con threads
  - Inicialización
    - MPI_Init_thread(int *argc, char **argv[], int required, int *provided)
    - Los nuevos argumentos indican el nivel de soporte solicitado y el proporcionado, respectivamente
      - MPI_THREAD_SINGLE (equivalente a MPI_Init), MPI_THREAD_/FUNNELED/SERIALIZED/MULTIPLE
  - Otras funciones
    - MPI_Query_thread(int *provided)
    - MPI_Is_thread_main(int *flag)

MPI-G2

- Implementaciones de MPI para Grid
- MPICH-G2 es la más utilizada
  - Implementación *Grid enabled* de MPI v1.1 standard
  - Utiliza los servicios proporcionados por Globus Toolkit
  - Gestiona los mensajes seleccionando el protocolo de comunicación
    - TCP entre diferentes máquinas
    - Proporcionado por el fabricante dentro de cada máquina
  - Los códigos paralelos MPI no cambian

Bibliografía

- http://www.mpi-forum.org/docs/docs.html
- W. Gropp, E. Lusk and A. Skjellum. *Using MPI.* The MIT Press, 1994. (1999 la 2ª edición)
- W. Gropp, E. Lusk and R. Thakur. *Using MPI-2.* The MIT Press, 1999.
- http://www.llnl.gov/computing/tutorials/mpi/
- MPICH: http://www.mcs.anl.gov/mpi
- LAM MPI: http://www.lam-mpi.org
- MPI-G2: http://www3.niu.edu/mpi/