Centro Nacional de Cálculo Científico
Universidad de Los Andes
CeCalCULA
Mérida - Venezuela

Introducción a *MPI* *(Message Passing Interface)*

Francisco Hidrobo
hidrobo@ciens.ula.ve

Herbert Hoeger
hhoeger@ing.ula.ve

CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

1. INTRODUCCION A *MPI*

1.1. GENERALIDADES

El *pase de mensajes* es un modelo de comunicación ampliamente usado en computación paralela. En años recientes se han logrado desarrollar aplicaciones importantes basadas en este paradigma. Dichas aplicaciones (incluyendo algunas del dominio público) han demostrado que es posible implementar sistemas basados en el pase de mensajes de una manera eficiente y portable. El crecimiento en el volumen y diversidad de tales aplicaciones originaron la necesidad de crear un estándar, es así como surge *MPI* o *Message Passing Interface*

MPI es un estándar para la implementación de sistemas de pase de mensajes diseñado por un grupo de investigadores de la industria y la academia para funcionar en una amplia variedad de computadores paralelos y de forma tal que los códigos sean portables. Su diseño esta inspirado en máquinas con una arquitectura de memoria distribuida en donde cada procesador es propietario de cierta memoria y la única forma de intercambiar información es a través de mensajes, sin embargo, hoy en día también encontramos implemetaciones de *MPI* en máquinas de memoria compartida. El estándar define la sintaxis y semántica de un conjunto de rutinas útiles para un amplio grupo de usuarios. Los diseñadores de *MPI* tomaron las características más atractivas de varios sistemas de pase de mensajes existentes, en vez de adoptar a uno de ellos como el estándar. Así, *MPI* ha sido fuertemente influenciado por trabajos en T.J. Watson Research Center (IBM), NX/2 (Intel), Express (Parasoft), Vertex (nCUBE), P4 (ANL), y PARMACS (ANL); además de otras contribuciones de Zipcode (MSU), Chimp (Edinburg University), PVM (ORNL, UTK, Emory U.), Chameleon (ANL) y PICL (ANL).

Desde que fue completado en Junio 1994, *MPI* ha ido creciendo tanto en aceptación como en uso. Se han realizado implementaciones en una gran variedad de plataformas que van desde máquinas masivamente paralelas hasta redes de estaciones de trabajo. Incluso, grandes compañías que fabrican Supercomputadores incluyen a *MPI* como una de sus herramientas fundamentales.

El objetivo principal de *MPI* es lograr la portabilidad a través de diferentes máquinas, tratando de obtener un grado de portabilidad comparable al de un lenguaje de programación que

permita ejecutar, de manera transparente, aplicaciones sobre sistemas heterogéneos; objetivo que no debe ser logrado a expensas del rendimiento.

1.2. EVOLUCION DEL ESTANDAR

El proceso de estandarización de *MPI* ha involucrado a más de 80 personas de 40 organizaciones, principalmente en los Estados Unidos y Europa. Este proceso comenzó en Abril de 1992 con el *Workshop on Standards for Message Passing in Distributed Memory Enviroment* patrocinado por el *Center for Research on Parallel Computing* en Williamsburg, Virginia. Una versión preliminar del estándar, conocida como *MPI1*, se presentó en Noviembre de ese mismo año, terminándose la revisión en Febrero de 1993.

Durante una reunión del grupo de *MPI* en Minneapolis, realizada en Noviembre de 1992, se decide formalizar el proceso de estandarización mediante la creación de un foro (*MPI Forum*), que luego se abriría a toda la comunidad de computación de alto rendimiento. En Noviembre de 1993, en Supercomputing'93, se presenta el estándar *MPI*. Luego de un periodo de comentarios y revisiones, que resultaron en algunos cambios a *MPI*, la versión 1.0 fue liberada el 5 de Mayo de 1994. Comenzando en Marzo de 1995, el foro de *MPI* convino en corregir errores y hacer aclaraciones de la versión 1.0. Tales discusiones resultaron en la versión 1.1 liberada en Junio de 1995.

1.3. OBJETIVOS DE *MPI*

De manera simple, el objetivo de *MPI* es desarrollar un estándar para ser ampliamente usado que permita escribir programas usando pase de mensajes. Por lo tanto, la interface debería establecer un estándar práctico, portable, eficiente y flexible.

A continuación se presentan los objetivos de *MPI*:

- Diseñar una API (Application Programming Interface).
- Hacer eficiente la comunicación.
- Permitir que las aplicaciones puedan usarse en ambientes heterogéneos.
- Soportar conexiones de la interface con C y Fortran 77, con una semántica independiente del lenguaje.
- Proveer una interface de comunicación confiable.

- No diferir significativamente de algunas implementaciones tales como PVM, p4, NX, express, etc., extendiendo la flexibilidad de éstas.
- Definir una interface implementable en plataformas de diferentes proveedores sin tener que hacer cambios significativos.

1.4. CARACTERISTICAS

- Generales:
  - Los comunicadores combinan procesamiento en contexto y por grupo para seguridad de los mensajes.
  - Seguridad en la manipulación de aplicaciones con hebrado.
- Manejo de ambiente. *MPI* incluye definiciones para:
  - Temporizadores y sincronizadores.
  - Inicializar y finalizar.
  - Control de errores.
  - Interacción con el ambiente de ejecución.
- Comunicación punto a punto:
  - Heterogeneidad para *buffers* estructurados y tipos de datos derivados.
  - Varios modos de comunicación:
    - Normal, con y sin bloqueo.
    - Síncrono.
    - Listo, para permitir acceso a protocolos rápidos.
    - Retardados, utilizando *buffers*.
- Comunicaciones colectivas:
  - Capacidad de manipulación de operaciones colectivas con operaciones propias o definidas por el usuario.
  - Gran número de rutinas para el movimiento de datos.
  - Los subgrupos pueden definirse directamente o por la topología.
- Topologías por procesos:
  - Soporte incluido para topologías virtuales para procesos (mallas y grafos).

- Caracterización de la Interface:
  - Se permite al usuario interceptar llamadas *MPI* para instalar sus propias herramientas.

1.5. IMPLEMENTACIONES

1.5.1. Del Dominio Público

Algunas de las implementaciones del dominio público de *MPI* son:

- **MPICH** (*MPI*/Chameleon). Producida por el Laboratorio Nacional de Argonne y la Universidad del Estado de Mississippi.
- **LAM** (Local Area Multicomputer) es un ambiente de programación y un sistema de desarrollo sobre redes de computadores heterogéneas. Esta implementación es del Centro de Supercomputación de Ohio.
- **CHIMP** (Common High-level Interface to Message Passing). Desarrollado por El Centro de Computación Paralela de Edinburgo.
- **UNIFY**. Provee un subconjunto de *MPI* dentro del ambiente PVM sin sacrificar las llamadas ya disponibles de PVM. Fue desarrollado por la Universidad del Estado de Mississipi. Corre sobre PVM y provee al programador de un uso de API dual; un mismo programa puede contener código de PVM y *MPI*.
- **MPICH/NT**. Es una implementación de *MPI* para una red de estaciones con Windows NT. Esta basada en MPICH y esta disponible de la Universidad del Estado de Mississipi.
- **W32MPI**. Implementación de *MPI* para un conglomerado con MS-Win32 basado también en MPICH y esta disponible del Instituto de Ingeniería Superior de Coimbra - Portugal y de la Universidad de Coimbra.
- **WinMPI**. WinMPI es la primera implementación de *MPI* para MS-Windows 3.1. Este corre sobre cualquier PC. Fue desarrollado por la Universidad de Nebraska en Omaha.
- ***MPI*-FM**. *MPI*-FM es un puerto de alto rendimiento de MPICH para un conglomerado de SPARCstation interconectadas por Myrinet. Esta basado en mensajes rápidos y viene de la Universidad de Illinois en Urbana-Champaign.

1.5.2. Información de otras implementaciones *MPI*

- Implementación de *MPI* de SGI.
- *MPI* para la Fujitsu AP1000. David Sitsky de la Universidad Nacional de Australia estuvo desarrollando una implementación de *MPI* para la Fujitsu AP1000.
- CRI/EPCC *MPI* para la Cray/T3D. El Centro de Computación Paralela de Edinburgo (EPCC) ha trabajado con Cray Research, Inc. (CRI) para desarrollarar una versión de *MPI* sobre la Cray T3D.
- MPIF para la IBM SP1/2. MPIF es una implementación eficiente de IBM para la IBM SP1/2.

Diferentes versiones de MPI e información sobre ellas pueden ser encontradas en http://www.mcs.anl.gov/Projects/mpi/implementations.html, http://www.mpi.nd.edu/MPI o en http://www.erc.msstate.edu/mpi/implementations.html.

1.6. TAMAÑO DE *MPI*

El estándar es extenso en cuanto al número de rutinas que se especifican; contiene alrededor de 129 funciones muchas de la cuales tienen númerosas variantes y parámetros. Para aprovechar las funcionalidades extendidas de *MPI* se requieren muchas funciones. Esto no implica una relación directamente proporcional entre la complejidad y el número de funciones; muchos programas paralelos pueden ser escritos usando sólo 6 funciones básicas. Sin embargo, *MPI* permite flexibilidad cuando sea requerida, además de que no se debe ser un experto en todas los aspectos de *MPI* para usarlo satisfactoriamente. En esta *Introducción a MPI* solo se cubrirá algunas de la funciones usualmente más usadas.

1.7. USO DE *MPI*

1.7.1. Cuando usar *MPI*

- Si la portabilidad es necesaria. Si se requiere una aplicación paralela portable, *MPI* será

una buena elección.

- En el desarrollo de bibliotecas paralelas.
- La interrelación de los datos es dinámica o irregular y no se ajusta a un modelo de datos paralelos. *MPI*, y en general el esquema de pase de mensajes, facilitan la programación en estos casos.

1.7.2. Cuando NO usar *MPI*

- Si es posible usar HPF (High Performance Fortran) o Fortran 90 paralelo.
- Es posible usar una biblioteca de más alto nivel (la cual podría estar escrita en *MPI*).

2. ELEMENTOS BÁSICOS DE LA PROGRAMACIÓN CON *MPI*

2.1. INTRODUCCIÓN

Uno de los objetivos de *MPI* es lograr portabilidad del código fuente. Por lo tanto, un programa escrito en *MPI* no debe requerir cambios en el código fuente cuando sea migrado de un sistema a otro siempre y cuando cumpla con los estandares del lenguaje base. Explícitamente el estándar *MPI* no especifica como un programa es arrancado o que debe hacer el usuario con su ambiente para lograr ejecutar un programa *MPI*. Sin embargo, una implementación puede requerir cierta configuración previa antes de poder ejecutar llamadas a rutinas de *MPI*. Para esto, *MPI* incluye una rutina de inicialización *MPI_Init* (Figura 1)

Paralelamente existe la rutina *MPI_Finalize* que permite purgar o eliminar todos los estados de *MPI*. Una vez que esta rutina es llamada ninguna otra rutina (ni siquiera *MPI_Init*) puede ser llamada. El usuario debe asegurarse de que todas las acciones pendientes sean completadas antes de llamar a *MPI_Finalize* (Figura 1).

MPI_Init()

Inicia el computo

(argc y argv solo son requeridos en C)

int MPI_Init(int *argc, char ***argv)

MPI_INIT(IERROR)
INTEGER IERROR

MPI_Finalize()

Finaliza el computo

int MPI_Finalize(void)

MPI_FINALIZE(IERROR)
INTEGER IERROR

Figura 1. Funciones para iniciar y finalizar *MPI*

```c
#include <stdio.h>
#include <mpi.h>
int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  printf("Hola Mundo!\n");
  MPI_Finalize();
}
```

```
program main
include 'mpif.h'
integer ierr

call MPI_INIT( ierr )
print *, 'Hola Mundo!'
call MPI_FINALIZE( ierr )
end
```

Figura 2. Programa Hola Mundo - Versión 1 en C y Fortran

En la Figura 2 se muestra un ejemplo muy sencillo escrito en C y Fortran respectivamente, en donde cada proceso imprime el mensaje: `Hola Mundo!`.Todas las rutinas *MPI* utilizan el prefijo *MPI_* En las rutinas de C, luego del prefijo se coloca el nombre de la rutina con la primera letra en mayúscula; mientras que para Fortran, el nombre va todo en mayúsculas y se utiliza la instrucción *call* para invocar las rutinas. Todas las rutinas en Fortran (excepto *MPI_WTIME* y *MPI_WTICK*) tienen el parámetro adicional *ierr*. `#include <mpi.h>` (`include 'mpif.h'` para Fortran) provee los tipos y las definiciones básicas para *MPI*. Esta instrucción es obligatoria en todo programa *MPI*. Los argumentos `argc` y `argv` en *MPI_Init* solo son requeridos en C. Finalmente, todas las llamadas a rutinas que no son *MPI* se ejecutan de manera local; de esta manera, el *printf* (*print* para Fortran) corre en cada proceso.

2.2. COMPILANDO Y ENLAZANDO PROGRAMAS *MPI*

Para un programa sencillo, un comando especial de compilación puede ser usado. Para proyectos grandes, es mejor un *Makefile* estándar.

Algunas implementaciones, como *MPICH* (la cual asumiremos de aquí en adelante) y *LAM*, proveen los comandos *mpicc* y *mpif77* para compilar programas en C y Fortran respectivamente.

2.2.1. Comandos de compilación

Para compilar, usualmente, se usan los comandos: *mpicc* y *mpif77* cuyo funcionamiento y parámetros son muy similares a los de los compiladores *cc* y *f77*.

```
mpicc -o holamundo holamundo.c   o   mpif77 -o holamundo holamundo.f
```

Figura 3. Compilando con *mpicc* y *mpif77*

2.2.2. Construcción del archivo Makefile

Además de los comandos de compilación, MPICH incluye una plantilla de ejemplo para la construcción de archivos *Makefile* denominada *Makefile.in*. Dicho archivo genera, a través de traducción, el archivo *Makefile*. Para lograr esta traducción debe utilazarse el programa (*script*) *mpireconfig* que construirá el archivo *Makefile* para un sistema particular.

Esto permite usar el mismo *Makefile* para una red de estaciones de trabajo o para un computador masivamente paralelo aunque utilicen distintos compiladores, bibliotecas y opciones de enlace.

```
mpireconfig Makefile
```

Figura 4. Creando el archivo *Makefile*

2.3. EJECUTANDO PROGRAMAS *MPI*

Tal como Fortran no especifica como serán arrancados los programas, *MPI* no especifica como se arrancaran los procesos *MPI*. Las implementaciones se encargaran de definir los mecanismos de arranque de procesos; para tales fines, algunas versiones coinciden en el uso de un programa especial denominado *mpirun*. Dicho programa no es parte del estándar *MPI*.

La opción *-t* muestra los comandos que *mpirun* ejecuta y puede ser usada para determinar como *mpirun* arranca los programas en un sistema determinado.

```
mpirun -np 2 -machinefile maquinas holamundo
```

Figura 5. Ejecutando el programa *holamundo*

En la Figura 5 el parámetro *np* le indica al programa *mpirun* que debe crear dos instancias de *holamundo*; mientras que *machinefile* indica el nombre del archivo (*maquinas*) que contiene la descripción de las máquinas que serán utilizadas para correr los procesos. Adicionalmente, *mpirun* tiene un conjunto de opciones que permiten al usuario ordenar la ejecución de un programa bajo determinadas condiciones. Estas opciones pueden verse con el comando:

```
man mpirun   o   mpirun -help
```

Figura 6. Obteniendo ayuda sobre *mpirun*

2.4. INFORMACIÓN DEL AMBIENTE

Algunas de las preguntas más importantes en todo programa paralelo son: ¿cuantos procesos hay?, ¿quien soy yo? y ¿en donde estoy corriendo?. El valor para el número de procesos se obtiene con la rutina *MPI_Comm_size*, *quien soy yo* con *MPI_Comm_rank* y *en donde estoy corriendo* con *MPI_Get_processor_name*. El identificador o rango es un número entre cero (0) y el total de procesos menos uno (procesos -1). Los parámetros se detallan en la Figura 7. En esta Figura y en las futuras se designará con E, S y E/S (Entrada, Salida, Entrada/Salida) para indicar si la función usa pero no modifica el parámetro (E), no lo usa pero lo modifica (S), o lo usa y lo modifica (E/S). Tambien se indica la declaración tanto en ANSI C como en Fortran 77. En C, todos los tipos que comiencen con MPI_ son tipos predefinidos de *MPI* que se encuentran en "*mpi.h*".

MPI maneja el sistema de memoria que es usado para almacenar mensajes y las representaciones internas de objetos de *MPI* como grupos, comunicadores, tipos de datos, etc. El usuario no puede accesar directamente esta memoria y los objetos ahí almacenados se dicen que son *opacos*: su tamaño y forma no son visibles al usuario. Los objetos *opacos* son accesados usando *asas* (*handles*), que existen en el espacio del usuario. En Fortran las *asas* son enteros, mientras que en C una *asa* diferente es definida por cada categoría.

Un comunicador identifica un grupo de procesos y el contexto en el cual se llevará a cabo una operación. Ellos proveen (como se verá más adelante) un mecanismo para identificar subconjuntos de procesos en el desarrollo de programas modulares y para garantizar que mensajes concebidos para diferentes propósitos no sean confundidos. El valor por omisión

MPI_COMM_WORLD identifica todos los procesos involucrados en un cómputo.

MPI_Comm_size(comm, nproc)

Retorna el número de procesos involucrados en un computo

E	comm	comunicador
S	nproc	número de procesos en el grupo de comm

int MPI_Comm_size(MPI_Comm comm, int *nproc)

MPI_COMM_SIZE(COMM, NPROC, IERROR)
INTEGER COMM, NPROC, IERROR

MPI_Comm_rank(comm, my_id)

Retorna el identificador o rango de un proceso

E	comm	comunicador
S	my_id	rango del proceso en el grupo de comm

int MPI_Comm_rank(MPI_Comm comm, int *my_id)

MPI_COMM_RANK(COMM, MY_ID, IERROR)
INTEGER COMM, MY_ID, IERROR

MPI_Get_processor_name(name, resultlen)

Retorna el nombre del procesador sobre el cual se esta corriendo al momento de hacer la llamada

S	name	nombre del procesador
S	resultlen	longitud en número de caracteres de *name*

int MPI_Get_processor_name(char *name, int *resultlen)

MPI_GET_PROCESSOR_NAME(NAME, RESULTLEN, IERROR)
CHARACTER*(*) NAME
INTEGER RESULTLEN, IERROR

Figura 7. Funciones para obtener información sobre el ambiente

En la Figura 8 se muestra una modificación al programa *holamundo* en donde cada proceso solicita su identificador y el número total de procesos que se están ejecutando. El código en C, además muestra como se usa la función *MPI_Get_processor_name*. Dado que el uso de *MPI* en C y Fortran es análogo, de aquí en adelante solo se darán ejemplos en C.

```
#include <stdio.h>
#include <mpi.h>
int my_id, nproc, resultlen;
char name[30];
int main (int argc, char** argv)
{
```

```
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  MPI_Get_processor_name(name, &resultlen);
  name[resultlen] = '\0';
  printf("Hola Mundo! Yo soy %d de %d corriendo en %s\n",my_id,nproc,name);
  MPI_Finalize();
}
```

```
program main
include 'mpif.h'
integer ierr, my_id, nproc

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, my_id, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, nproc, ierr )
print *, 'Hola Mundo! Yo soy ', my_id, ' de ', nproc
call MPI_FINALIZE( ierr )
end
```

Figura 8. Programa Hola Mundo - Versión 2 en C y Fortran

2.5. MIDIENDO EL TIEMPO

Medir el tiempo que dura un programa es importante para establecer su eficiencia y para fines de depuración, sobre todo cuando éste es paralelo. Para esto *MPI* ofrece las funciones *MPI_Wtime* y *MPI_Wtick* (Figura 9). Ambas proveen alta resolución y poco costo comparadas con otras funciones similares existentes en POSIX, Fortran, etc.

MPI_Wtime devuelve un punto flotante que representa el número de segundos transcurridos a partir de cierto tiempo pasado, el cual se garantiza que no cambia durante la vida del proceso. Es responsabilidad del usuario hacer la conversión de segundos a otras unidades de tiempo: horas, minutos, etc.

MPI_Wtick permite saber cuantos segundos hay entre *tics* sucesivos del reloj. Por ejemplo, si el reloj esta implementado en hardware como un contador que se incrementa cada milisegundo, entonces *MPI_Wtick* debe devolver 10^{-3}.

MPI_Wtime()

Retorna el número de segundos transcurridos a partir de cierto tiempo pasado

double MPI_Wtime(void)

DOUBLE PRECISION MPI_WTIME()

MPI_Wtick()
Retorna la resolución de MPI_Wtime en segundos

double MPI_Wtick(void)
DOUBLE PRECISION MPI_WTICK()

Figura 9. Funciones para medir el tiempo.

Para usar *MPI_Wtime* se debe escribir un código como el que se muestra a continuación:

```
...
double tiempo_inicial, tiempo_final;
...
tiempo_inicial = MPI_Wtime();
... codigo al que se le medira el tiempo ...
tiempo_final = MPI_Wtime();
printf("Codigo ejecutado en %f segundos\n",tiempo_final – tiempo_inicial);
...
```

Figura 10. Ejemplo de como medir el tiempo

2.6. COMUNICACIÓN PUNTO A PUNTO

Lo que se conoce, en *MPI*, como comunicación punto a punto es el mecanismo básico de transferencia de mensajes entre un par de procesos, uno enviando, y el otro, recibiendo. La mayoría de los constructos de *MPI* están construidos alrededor de operaciones punto a punto; por lo tanto, esta sección es fundamental para el aprendizaje y uso de *MPI*.

MPI provee de un conjunto de funciones de envío y recepción de mensajes que permiten la comunicación de datos de cierto tipo con una etiqueta (*tag*) asociada. El tipo de datos del contenido de los mensajes se hace necesario para dar soporte a la heterogeneidad. La información del tipo es utilizada para realizar las conversiones en la representación cuando se envían datos de una arquitectura a otra. Mientras que, la etiqueta del mensaje permite, del lado del receptor, seleccionar un mensaje en particular si así lo desea.

2.6.1 Envío y recepción de mensajes

El envío y recepción de un mensaje involucra el intercambio de información entre procesos. Para proveer un mecanismo de transferencia de mensajes debe haber alguna manera de dar respuesta a las siguientes preguntas: ¿a quien se envían los datos?, ¿qué es lo que se envía? y ¿como hace el receptor para identificar el mensaje?

Las aspectos que *MPI* provee para el envío de datos y ganar portabilidad son los siguientes:

- Especificación de los datos a enviar con tres campos: dirección de inicio, tipo de dato y contador, donde tipo de dato puede ser:
  - Elemental: Todos los tipos de datos de C y Fortran (Tablas 1 y 2).
  - Arreglos continuos de tipos de datos.
  - Bloques dispersos de tipos de datos.
  - Arreglos indexados de bloques de tipos de datos.
  - Estructuras generales
- Los tipos de datos son construidos recursivamente.
- Las especificaciones de tipos de datos permiten comunicaciones heterogéneas.
- Eliminación de la longitud en favor del contador.

Tipo de dato *MPI*	**Tipo de dato C**
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	*unsigned int*
MPI_UNSIGNED_LONG	*unsigned long int*
MPI_FLOAT	*float*
MPI_DOUBLE	*double*
MPI_LONG_DOUBLE	*long double*
MPI_BYTE	
MPI_PACKED	

Tabla 1. Equivalencias de tipos de datos entre *MPI* y C

Los tipos de datos *MPI_BYTE* y *MPI_PACKED* no corresponden a ningún tipo de dato de C ni de Fortran. El tipo *MPI_BYTE* consiste de un byte (8 dígitos binarios). El uso y utilidad del tipo *MPI_PACKED* será tratado en el empaquetamiento de datos.

Tipo de dato *MPI*	**Tipo de dato Fortran**
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*

Tabla 2. Equivalencias de tipos de datos entre MPI y Fortran

Las funciones básicas de *MPI* para el envío y recepción bloqueado son *MPI_Send* y *MPI_Recv* respectivamente (Figura 11). En C, *status* es una estructura del tipo *MPI_Status* que contiene tres campos denominados *MPI_SOURCE*, *MPI_TAG* y *MPI_ERROR*; y puede contener campos adicionales. Así, *status.MPI_SOURCE*, *status.MPI_TAG* y *status.MPI_ERROR* contienen el enviador, la etiqueta y el código de error, respectivamente, del mensaje recibido. En el caso de Fortran, *status* es un arreglo de enteros y estos valores estarán almacenados en *status(MPI_SOURCE*), *status(MPI_TAG)* y *status(MPI_ERROR).*

En las Figuras 12 y 13 se presentan dos ejemplos para ilustrar el uso de *MPI_Send* y *MPI_Recv*. En el primero de ellos se tiene un conjunto de procesos númerados 0, 1, ..., nproc-1 y cada proceso le envia a su sucesor su identificador (el sucesor de 1 es 2, y de nproc-1 es 0).

La operación `(my_id + 1)%nproc` retorna el residuo de la división de `(my_id + 1)` por `nproc`; por ejemplo, si `my_id` es 3 y `nproc` es 5, `(my_id + 1)%nproc` retorna `4`. En este ejemplo es indistinto usar las variables *MPI_ANY_SOURCE* (reciba de cualquier fuente) y *MPI_ANY_TAG* (cualquier etiqueta) o especificar la fuente y la etiqueta como se hace con la función que esta comentada (entre /* y */); no hay ambigüedad. Sin embargo, es bueno resaltar que el orden en que aparecen las funciones *MPI_Send* y *MPI_Recv* (necesariamente *MPI_Recv* debe ir después de *MPI_Send*) es imprescindible ya que *MPI_Recv* por ser bloqueante, no procede hasta haber recibido un mensaje. De invertir el orden, todos los procesos se quedarían esperando por un mensaje y ninguno puede proceder a enviarlo.

MPI_Send(buf, count, datatype, dest, tag, comm)

Envia un mensaje

E	buff	dirección del *buffer* de envio
E	count	número de elementos a enviar
E	datatype	tipo de datos de los elementos del *buffer* de envio
E	dest	identificador del proceso destino
E	tag	etiqueta del mensaje
E	comm	comunicador

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

```
MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
    <tipo> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

MPI_Recv(buf, count, datatype, source, tag, comm, status)

Recivea un mensaje

S	buff	dirección del *buffer* de recepción
E	count	número de elementos a recibir
E	datatype	tipo de datos de los elementos del *buffer* de recepción
E	source	identificador del proceso enviador o *MPI_ANY_SOURCE*
E	tag	etiqueta del mensaje o *MPI_ANY_TAG*
E	comm	comunicador
S	status	objeto status

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
             MPI_Status status)
MPI_RECV(BUF, COUNT, DATATYPE, DEST, TAG, COMM, STATUS, IERROR)
    <tipo> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM,  STATUS(MPI_STATUS_SIZE), IERROR
```

Figura 11. Funciones básicas de envío y recepción de mensajes

En el segundo ejemplo, se efectúa la sumatoria de los primeros *n* enteros. Se puede demostrar que $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ y por lo tanto es fácil verificar que el resultado sea correcto. Cada proceso que interviene en el cómputo sumará parte de estos números.

El proceso de mayor rango (nproc - 1), es el encargado de llenar un vector *a* con los primeros *n* enteros, distribuir partes de el a sus colegas y finalmente recolectar las sumas parciales para obtener la suma total. En caso de que no sea posible obtener partes equitativas para cada proceso, el proceso de mayor rango (nproc - 1) tendrá un poco mas de trabajo y sumará también el resto (Figura 13).

```
#include <stdio.h>
#include <mpi.h>
int my_id, nproc, tag = 99, source;
MPI_Status status;
int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
```

```
  MPI_Send(&my_id,1,MPI_INT,(my_id+1)%nproc,tag,MPI_COMM_WORLD);
  MPI_Recv(&source,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
  /* MPI_Recv(&source,1,MPI_INT,(my_id+nproc-1)%nproc,tag,
              MPI_COMM_WORLD,&status); */
  printf("%d recibio mensaje de %d\n",my_id,source);
  MPI_Finalize();
}
```

Figura 12. Programa saluda al sucesor

Los programas que usan el modelo de pase de mensajes por lo general son no deterministicos: el orden de llegada de los mensajes enviados por dos procesos *p* y *q*, a un tercer proceso *r*, no esta definido. Sin embargo, *MPI* garantiza que los mensajes enviados de un proceso *p* a otro proceso *q* llegan en el orden que fueron enviados.

Hasta ahora, se ha podido observar la sencillez de *MPI* en donde es posible construir un amplia variedad de programas usando solo estas seis funciones: *MPI_Init, MPI_Finalize, MPI_Comm_rank, MPI_Comm_size, MPI_Send y MPI_Recv.*

3. COMUNICACIONES COLECTIVAS

En ciertas aplicaciones se requiere enviar un conjunto de datos a múltiples procesos o recibir en un único proceso datos de varios remitentes. Las comunicaciones colectivas permiten la transferencia de datos entre todos los procesos que pertenecen a un grupo específico (el tratamiento de los grupos se estudiará posteriormente). En este caso, no se usan etiquetas para los mensajes, estas se sustituyen por comunicadores.

```
#include <stdio.h>
#include <mpi.h>
#define size 10000
int my_id, nproc, i, tag  = 99, first, chunksize, flag;
int a[size];
double sum,psum;
MPI_Status status;
int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  /* Calcula el tamaño de las partes a enviar a los otros */
  chunksize = size/nproc;
  if (my_id == nproc - 1) {
    /* Rellena el vecto a */
    for (i=0;i<size;i++) a[i] = i + 1;
    /* Envia las partes a los otros */
    for (i=0;i<nproc-1; i++) {
```

```
      first = i*chunksize;
      MPI_Send(&a[first],chunksize,MPI_INT,i,tag,MPI_COMM_WORLD);
    }
    /* determina lo que debo sumar yo (nproc-1) */
    first = my_id*chunksize;
    chunksize = size - 1;
    /* efectua mi suma parcial */
    sum = 0;
    for (i=first;i<=chunksize;i++) sum = sum + a[i];
    /* obtener las sumas parciales y calcula la total */
    for (i=0;i<nproc-1; i++) {
      MPI_Recv(&psum,1,MPI_DOUBLE,MPI_ANY_SOURCE,tag,MPI_COMM_WORLD,&status);
      printf("Respondio %d - envio %10.0f\n",status.MPI_SOURCE,psum);
      sum = sum + psum;
    }
    printf("Resultado de la suma = %10.0f\n",sum);
  }
  else {
    /* recibe mi parte */
    MPI_Recv(a,chunksize,MPI_INT,nproc-1,tag,MPI_COMM_WORLD,&status);
    /* efectua mi suma parcial */
    psum = 0;
    for (i=0;i<chunksize;i++) psum = psum + a[i];
    /* devuelve mi suma parcial a (nproc - 1) */
    MPI_Send(&psum,1,MPI_DOUBLE,nproc-1,tag,MPI_COMM_WORLD);
  }
  MPI_Finalize();
}
```

Figura 13. Programa paralelo para sumar los primeros *n* enteros

Las operaciones colectivas de *MPI* pueden ser clasificadas en tres clases:

- Sincronización. Barreras para sincronizar.
- Movimiento (transferencia) de datos. Operaciones para difundir, recolectar y esparcir.
- Cálculos colectivos. Operaciones para reducción global, tales como suma, máximo, mínimo o cualquier función definida por el usuario.

Estas operaciones estan resumidas en las Figura 14, 15 y 16. Todas las funciones, excepto la de difusión, tienen dos variantes: una en la cual todos los datos transferidos son del mismo tamaño (*homogéneo*) y otra donde cada *ítem* de datos puede tener un tamaño distinto (*vector*). Algunas de estas funciones, como la de difusión y recolección, tienen un proceso originador o un proceso receptor. Dicho proceso es llamado *raíz* (*root*).

MPI_Barrier(comm)

Sincronización global

E	comm	comunicador

```
int MPI_Barrier(MPI_Comm comm)
```

```
MPI_BARRIER(COMM, IERROR)
    INTEGER COMM, IERROR
```

MPI_Bcast(inbuf, incnt, intype, root, comm)

Difunde datos desde root a todos los procesos

ES	inbuf	dirección del *buffer* de recepción, o *buffer* de envio en *root*
E	incnt	número de elementos en el *buffer* de envio
E	intype	tipo de dato de los elementos del *buffer* de envio
E	root	rango del proceso *root*
E	comm	comunicador

```
int MPI_Bcast(void *inbuf, int incnt, MPI_Datatype intype, int root, MPI_Comm comm)
MPI_BCAST(INBUF, INCNT, INTYPE, ROOT, COMM, IERROR)
    <tipo> INBUF(*)
    INTEGER INCNT, INTYPE, ROOT, COMM, IERROR
```

Figura 14. Funciones colectivas

MPI_Barrier detiene la ejecución de cada proceso que la llama hasta tanto todos los procesos incluidos en el grupo asociado a *comm* la hayan llamado. *MPI_Bcast* difunde un mensaje del proceso identificado como *root* a todos los procesos del grupo. *MPI_Gather* hace que cada proceso, incluyendo al *root*, envie el contenido de su *buffer* de envio (*inbuf*) al proceso *root*. El proceso *root* recibe los mensajes y los almacena en orden (según el rango del enviador) en el *buffer* de recepción (*outbuf*). *MPI_Scatter* es la operación inversa al *MPI_Gather*. El proceso *root* enviará a cada proceso un trozo del contenido del *buffer* de envío, comenzando desde de la dirección inicial de dicho *buffer* se desplazara una cantidad (*incount*) para realizar el siguiente envio. *MPI_Reduce c*ombina los elementos provisto en el *buffer* de entrada (*inbuf*) de cada proceso en el grupo, usando la operación *op* (puede ser una operación predefinida o definida por el usuario) y retorna el valor combinado en el *buffer* de salida (*outbuf*) del proceso *root*. El orden de evaluación canónico de una reducción esta determinado por el rango de los procesos. Si se usa *MPI_Allreduce* se retorna el valor en los *buffers* de salida de cada uno de los procesos que intervienen. Las operaciones predefinidas validas de *MPI* se dan en la Tabla 3.

Nombre *MPI*	**Operación**
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_PROD	Producto

MPI_SUM	Suma
MPI_LAND	Y lógico
MPI_LOR	O lógico
MPI_LXOR	O exclusivo lógico
MPI_BAND	Y bit a bit
MPI_BOR	O bit a bit
MPI_BXOR	XOR bit a bit
MPI_MAXLOC	Máximo y localización
MPI_MINLOC	Mínimo y localización

Tabla 3. Operaciones validas en *MPI_Reduce*

Adicionalmente, *MPI* provee variantes de estas rutinas: *Alltoall, Alltoallv, Gatherv, Allgather, Scatterv, Allreduce, Scan, ReduceScatter.* Todas las versiones envían el resultado de la operación a los procesos participantes. Las versiones con el sufijo *v* permiten que los *trozos* a transferir tengan distintos tamaños. Para ilustrar estas operaciones se presentan dos modificaciones al programa de suma de los primeros *n* enteros. El primer programa (Figura 17) hace uso de *MPI_Bcast* y *MPI_Reduce*. El proceso con rango 0 rellena el vector *a* y lo difunde a los otros procesos (todo el vector). Cada proceso computa su respectiva suma parcial y por último se usa *MPI_Reduce* para obtener las sumas parciales y calcular la suma total. Notese que todos los procesos involucrados deben llamar a *MPI_Bcast* y *MPI_Reduce*, pero especificando *root* se sabe quien difunde (y quienes reciben) y quien reduce.

MPI_Gather(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

MPI_Scatter(inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

Funciones de tranferencia de datos colectivas

ES	inbuf	dirección del *buffer* de entrada
E	incnt	número de elementos a enviar a cada uno
E	intype	tipo de dato de los elementos del *buffer* de entrada
S	outbuf	dirección del *buffer* de salida
E	outcnt	número de elementos a recibir de cada uno
E	outtype	tipo de dato de los elementos del *buffer* de salida
E	root	rango del proceso *root*
E	comm	comunicador

```
int MPI_Gather(void *inbuf, int incnt, MPI_Datatype intype,
               void *outbuf, int outcnt, MPI_Datatype outtype, int root, MPI_Comm comm)
```

```
MPI_GATHER(INBUF, INCNT, INTYPE, OUTBUF, OUTCNT, OUTTYPE, ROOT,
           COMM, IERROR)
    <tipo> INBUF(*), OUFBUF(*)
    INTEGER INCNT, INTYPE, OUTCNT, OUTTYPE, ROOT, COMM, IERROR

int MPI_Scatter(void *inbuf, int incnt, MPI_Datatype intype,
                void *outbuf, int outcnt, MPI_Datatype outtype, int root, MPI_Comm comm)
MPI_SCATTER(INBUF, INCNT, INTYPE, OUTBUF, OUTCNT, OUTTYPE, ROOT,
           COMM, IERROR)
    <tipo> INBUF(*), OUFBUF(*)
    INTEGER INCNT, INTYPE, OUTCNT, OUTTYPE, ROOT, COMM, IERROR
```

Figura 15. Funciones colectivas (continuación)

El programa de la Figura 18 usa *MPI_Scatter* en vez de *MPI_Bcast* para difundir los datos respectivos a los procesos para que efectúen las sumas parciales. Al igual que en el programa anterior, se usa *MPI_Reduce* para obtener la suma total. Dado que *MPI_Scatter* distribuye el mismo número de elementos a cada proceso, se requiere que *size* sea divisible por *nproc*, en caso contrario habrían algunos números que quedarían fuera del cómputo. Nótese que los datos se distribuyen en forma ordenada; es decir, el proceso 0 recibe el primer trozo, el 1 el segundo y así sucesivamente. Finalmente para ilustrar *MPI_Gather*, se presenta un programa (Figura 19) en donde el llenado del vector `a` se hace en forma distribuida. Cada proceso involucrado se encarga de llenar su parte respectiva y con *MPI_Gather* se reciben en forma ordenada los trozos. El proceso 0 imprime el vector `a`.

MPI_Reduce(inbuf, outbuf, count, type, op, root, comm)
MPI_Allreduce(inbuf, outbuf, count, type, op, root, comm)
Funciones de reducción colectivas

E	inbuf	dirección del *buffer* de entrada
S	outbuf	dirección del *buffer* de salida
E	count	número de elementos en el *buffer* de entrada
E	type	tipo de dato de los elementos del *buffer* de entrada
E	op	operación (ver Tabla 3).
E	root	rango del proceso *root*
E	comm	comunicador

```
int MPI_Reduce(void *inbuf, void *outbuf, int count, MPI_Datatype type, MPI_Op op, int root,
               MPI_Comm comm)
```

```
MPI_REDUCE(INBUF, OUTBUF, COUNT, TYPE, OP, ROOT, COMM, IERROR)
    <tipo> INBUF(*), OUTBUF(*)
     INTEGER COUNT, TYPE, OP, ROOT, COMM, IERROR

int MPI_Allreduce(void *inbuf, void *outbuf, int count, MPI_Datatype type, MPI_Op op, int root,
                  MPI_Comm comm)
MPI_ALLREDUCE(INBUF, OUTBUF, COUNT, TYPE, OP, ROOT, COMM, IERROR)
    <tipo> INBUF(*), OUTBUF(*)
     INTEGER COUNT, TYPE, OP, ROOT, COMM, IERROR
```

Figura 16. Funciones colectivas (continuación)

```
#include <stdio.h>
#include <mpi.h>
#define size 10000
int my_id, nproc, a[size], i, tag = 99, first, last, chunksize;
double sum = 0,psum = 0;
MPI_Status status;
int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  /* Rellena el vector a */
  if (my_id == 0) for (i=0;i<size;i++) a[i] = i + 1;
  /* Difunde el vector a */
  MPI_Bcast(a,size,MPI_INT,0,MPI_COMM_WORLD);
  /* Determina números a sumar */
  chunksize = size/nproc;
  if (my_id == nproc - 1) {

    first = (nproc - 1)*chunksize;
    last = size - 1;
    }
  else  {
    first = my_id*chunksize;
    last = (my_id + 1)*chunksize - 1;
  }
  /* Computa la suma parcial */
  for (i=first;i<=last;i++) psum = psum + a[i];
  /* Hacer la reducción de las sumas parciales para obtener la suma total */
  MPI_Reduce(&psum,&sum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
  if (my_id == 0) printf("Suma: %10.0f\n",sum);
  MPI_Finalize();
}
```

Figura 17. Suma de los *n* primeros enteros usando *MPI_Bcast* y *MPI_Reduce*

```
#include <stdio.h>
#include <mpi.h>
#define size 10000
```

```
int my_id, nproc, i, tag  = 99, first, chunksize;
int a[size], b[size];
double psum, sum;
MPI_Status status;
/*****************************************/
/* OJO: size debe ser divisible por nproc */
/*****************************************/
int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  chunksize = size/nproc;
  /* No calcule si size no es divisible por nproc */
  if (chunksize*nproc != size) {
    if (my_id == 0)
      printf("'size(%d)' debe ser divisible por 'nproc(%d)'\n",size,nproc);
    MPI_Finalize();
    exit(1);
  }
  /* Rellena el vector */
  if (my_id == 0) for (i=0;i<size;i++) a[i] = i + 1;
  /* Distribuye el vector */
  MPI_Scatter(a,chunksize,MPI_INT,b,chunksize,MPI_INT,0,MPI_COMM_WORLD);
  /* Calculen las sumas parciales */
  psum = 0;
  for (i=0;i<chunksize;i++) psum = psum + b[i];
  /* Hacer la reducción de las sumas parciales para obtener la suma total */
  MPI_Reduce(&psum,&sum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
  if (my_id == 0) printf("Suma: %10.0f\n",sum);
  MPI_Finalize();
}
```

Figura 18. Suma de los primeros *n* enteros usando *MPI_Scatter* y *MPI_Reduce*

```
#include <stdio.h>
#include <mpi.h>
#define size 1000
int my_id, nproc, i, tag  = 99, first, last, chunksize;
int a[size];
/*****************************************/
/* OJO: size debe ser divisible por nproc */
/*****************************************/
int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);
  chunksize = size/nproc;
  /* No calcule si size no es divisible por nproc */
  if (chunksize*nproc != size) {
    if (my_id == 0)
      printf("'size(%d)' debe ser divisible por 'nproc(%d)'\n",size,nproc);
    MPI_Finalize();
    exit(1);
  }
  /* Determinen y llenen el trozo respectivo - Esto es en forma ordenada */
```

```
  first = my_id*chunksize;
  last = (my_id+1)*chunksize;
  for (i=first;i<last;i++) a[i-first] = i + 1;
  /* Obtener los trozos de los otros procesos - Esto es en forma ordenada */
  MPI_Gather(a,chunksize,MPI_INT,a,chunksize,MPI_INT,0,MPI_COMM_WORLD);
  /* Imprimir el vector */
  if (my_id == 0){
    for (i=0;i<size;i++) {
      if (i%10 == 0) printf("\n");
      printf("%4d ",a[i]);
    }
    printf("\n");
  }
  MPI_Finalize();
}
```

Figura 19. Llenado distribuido del vector *a*

3.1. DEFINIENDO SUS PROPIAS OPERACIONES COLECTIVAS

MPI_Op_create (Figura 20) le permite al usuario enlazar una operación definida por él con una asa que puede ser usado en las rutinas *MPI_Reduce, MPI_Allreduce, MPI_Scan* y *MPI_Reduce_Scatter*. La operación definida por el usuario se asume que es asociativa. Si *commute = true,* entonces se asume asociativa y conmutativa. Son raras las ocasiones en que esto haga falta ya que la mayoría de las necesidades están cubiertas con las operaciones predefinidas en *MPI*. En la Figura 21 se presenta un ejemplo de como definir sus propias operaciones.

MPI_Op_create(function, commute, op)

Para definir funciones del usuario

E	function	función definida por el usuario
E	commute	cierto si la función es conmutativa
S	op	operación

int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)

MPI_OP_CREATE(FUNCTION, COMMUTE, OP, IERROR)
EXTERNAL FUNCTION
LOGICAL COMMUTE
INTEGER OP, IERROR

MPI_Op_free(op)

Destruye la definición de la función

E	op	operación

```
int MPI_Op_free(MPI_Op *op)
MPI_OP_FREE(OP, IERROR)
    INTEGER OP, IERROR
```

Figura 20. Funciones para definir operaciones del usuario

```
#include <stdio.h>
#include <mpi.h>
#define size 1000

typedef struct {
  double real, imag;
} complex;

void my_Prod();
int my_id, i;
complex a[size], sol[size];
MPI_Op my_op;
MPI_Datatype ctype;

int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

  MPI_Type_contiguous(2, MPI_DOUBLE, &ctype);
  MPI_Type_commit(&ctype);

  /* cada uno tiene su arreglo - ejemplo con datos artificiales */
  for (i = 0; i < size; i++) {
    a[i].real = (my_id + 1)*i;
    a[i].imag = (my_id + 1)*(size - i);
  }

  MPI_Op_create(my_Prod, 1, &my_op);
  MPI_Reduce(a, sol, size, ctype, my_op, 0, MPI_COMM_WORLD);

  /* el proceso root tiene la soluci□n */
  if (my_id == 0)
    for (i = 0; i < size; i++)
      printf("(%10.0f,%10.0f)\n",sol[i].real,sol[i].imag);

  MPI_Op_free(&my_op);
  MPI_Finalize();
}

void my_Prod(complex *in, complex *inout, int *len, MPI_Datatype *dptr)
{
  int i;
  complex c;
  for (i = 0; i < *len; ++i) {
    c.real = inout->real*in->real - inout->imag*in->imag;
    c.imag = inout->real*in->real + inout->imag*in->imag;
    *inout = c;
    in++; inout++;
```

```
}
```

Figura 21. Ejemplo de definición de operaciones del usuario

La funcion definida por el usuario debe seguir los siguientes prototipos de C y Fortran respectivamente:

```
typedef void MPI_User_function(void *in, void *inout, int *len, MPI_Datatype *type)
FUNCTION USER_FUNCTION( IN(*), INOUT(*), LEN, TYPE)
  <type> IN(LEN), INOUT(LEN)
  INTEGER LEN, TYPE
```

`type` es un asa al tipo de datos que se pasa en la llamada a *MPI_Reduce*; `ctype` en el ejemplo, e informalmente podemos entender que `in` e `inout` son vectores de longitud `len` con elementos de tipo `ctype`, en donde la función de reducción re-escribe los valores de `inout` tal que para cada invocación de la función tenemos que `inout[i] = in[i]`☜`inout[i]`, para `i = 0,...,len-1`, en donde ☜ es la operación computada por la función.

4. TRANSFERENCIA DE DATOS DISPERSOS O NO HOMOGENEOS

Los ejemplos y ejercicios presentados, hasta el momento, involucran el envío y recepción de secuencias de elementos idénticos y continuos en memoria. Sin embargo, en algunas aplicaciones es necesario transferir datos no homogéneos como las estructuras (registros) o que no están continuos en memoria como secciones de arreglos. Lo ideal es tener un mecanismo que permita estas transferencias sin necesidad de utilizar grandes cantidades de mensajes. Para lograr esto, *MPI* provee dos esquemas:

1. El usuario puede definir tipos de datos *derivados,* que especifiquen diseños de datos generales. Esos nuevos tipos de datos pueden ser usados en las funciones de comunicación de *MPI* en vez de los tipos de datos básicos predefinidos.
2. El proceso que envía puede explícitamente empaquetar datos no continuos en un *buffer* continuo y luego enviarlo; mientras que el proceso receptor desempaqueta los datos de un *buffer* continuo para almacenarlos en localizaciones no continuas.

4.1 DEFINICION DE TIPOS DE DATOS EN *MPI*

MPI tiene una gran potencialidad al permitir al usuario construir nuevos tipos de datos en

tiempo de ejecución; estos nuevos tipos son llamados tipos de datos derivados. A través de los tipos de datos derivados, *MPI* soporta la comunicación de estructuras de datos complejas tales como secciones de arreglos y estructuras que contienen combinaciones de tipos de datos primitivos. Los *tipos derivados* se construyen a partir de tipos de datos básicos usando los constructos que se muestran en la Figuras 23 y 24. Un *tipo derivado* es un objeto *opaco* que especifica dos cosas: una secuencia de datos primitivos y una secuencia de desplazamientos.

La extensión de un tipo de dato se define como el espacio que hay del primer byte al último byte que ocupa el tipo de dato, redondeado hacia arriba para satisfacer requerimientos de alineación. Si por ejemplo *tipo* = {(double,0),(char,8)}, un doble con desplazamiento 0 y un carácter con desplazamiento 8, y los dobles deben ser alineados en direcciones que son múltiplos de 8, entonces el tamaño de este tipo es 9 mientras que su extensión es 16 (9 redondeado al próximo múltiplo de 8). La extensión y tamaño de un tipo de dato se puede obtener con *MPI_Type_extent* y MPI_Type_size respectivamente (Figura 22).

Los constructos que se presentan en la Figura 23 y 24 permiten definir tipos derivados y se presentan en orden de más simple a más complejo. *MPI_Type_contiguous* es el constructo para tipos más simple. Define un arreglo de tipos de datos iguales que se agrupan de forma continua; *newtype* se obtiene concatenando *count* copias de *oldtype*.

oldtype 

count = 4

newtype 

MPI_Type_extent(datatype, extent)

Para obtener la extensión de un tipo de dato

ES	datatype	tipo de dato
S	extent	extensión del tipo de dato

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
MPI_TYPE_EXTENT(MPI_Type_extent(datatype, extent))
    INTEGER MPI_Type_extent(datatype, extent)
```

MPI_Type_size(datatype, size)

Para obtener la extensión de un tipo de dato

ES	datatype	tipo de dato
S	size	tamaño del tipo de dato

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
    INTEGER DATATYPE, SIZE, IERROR
```

Figura 22. Funciones para obtener la extensión y el tamaño de un tipo de dato

MPI_Type_vector define un tipo de dato derivado que consiste de un arreglo de bloques separados por una distancia fija. Cada bloque se obtiene por la concatenación de un mismo n*úmero* de copias del tipo de dato base (oldtype). El espacio entre bloques es un multiplo de la extensión del tipo de dato viejo. MPI_Type_contiguous(cnt,oldtype,newtype) es equivalente a MPI_Type_vector(cnt,1,1,oldtype,newtype) o MPI_Type_vector(1,cnt,num,oldtype,newtype) con *num* arbitrario.

oldtype 

count = 3, blocklength = 2, stride = 3

newtype 

MPI_Type_contiguous(count, oldtype, newtype)

Definir un arreglo de datos

E	count	número de elementos en el arreglo
E	oldtype	tipo de dato viejo
S	newtype	tipo de dato nuevo

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
    INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR
```

MPI_Type_vector(count, blocklength, stride, oldtype, newtype)

Permite la replicación de datos hacia bloques igualmente espaciados

E	count	número de bloques
E	blocklength	número de elementos en cada bloque
E	stride	espacio entre el comienzo de cada bloque medido en número de elementos
E	oldtype	tipo de dato viejo
S	newtype	tipo de dato nuevo

```
MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI_Type_indexed(count, array_of_blocklength, array_of_displacements, oldtype, newtype)

Permite la replicación de datos hacia secuencias de bloques

E	count	número de bloques
E	array_of_blocklength	número de elementos por bloque
E	array_of_displacements	desplazamiento de cada bloque medido en número de elementos
E	oldtype	tipo de dato viejo
S	newtype	tipo de dato nuevo

int MPI_Type_indexed(int count, int *array_of_blocklength, int *array_of_displacements,
MPI_Datattype oldtype, MPI_Datatype *newtype)

MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTH, ARRAY_OF_DISPLACEMENTS,
OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTH(*), ARRAY_OF_DISPLACEMENTS(*),
OLDTYPE, NEWTYPE, IERROR

Figura 23. Funciones para definir tipos de datos derivados

MPI_Type_struct(count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

Permite la replicación de datos hacia bloques de diferentes tipos

E	count	número de bloques
E	array_of_blocklengths	número de elementos por bloque
E	array_of_displacements	desplazamiento en *bytes* de cada bloque
E	array_of_types	tipo de los elementos en cada bloque
S	newtype	tipo de dato nuevo

MPI_Type_struct(int count, int *array_of_blocklength, MPI_Aint *array_of_displacements,
MPI_Datatype *array_of_types, MPI_Datatype *newtype)

MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS, ARRAY_OF_DISPLACEMENTS,
ARRAY_OF_TYPES, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_DISPLACEMENTS(*),
ARRAY_OF_TYPES(*), NEWTYPE, IERROR

MPI_Type_commit(datatype)

Para cometer un tipo de datos

ES	datatype	tipo de dato a ser cometido

int MPI_Type_commit(MPI_Datatype *datatype)

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
    INTEGER DATATYPE, IERROR
```

Figura 24. Funciones para definir tipos de datos derivados (continuación)

En la Figura 25 hay un programa en donde se define un tipo derivado para la diagonal de una matriz. El proceso 0 rellena la matriz y luego envía a los otros procesos (1,...,*nproc*-1) sólo la diagonal de dicha matriz, la cual es recibida como un vector.

El programa de la Figura 26 define un tipo de dato derivado que representa la matriz traspuesta de una matriz *size*x*size*. *MPI_Type_hvector* es análogo a *MPI_Type_vector* pero *stride* se especifica en número de *bytes* y no en número de elementos. En este programa el proceso 0 rellena la matriz y al enviarla la traspone. Los receptores, procesos 1,...,*nproc*-1, reciben la matriz ya traspuesta. El mismo propósito se lograría enviando la matriz sin trasponer, en cuyo caso el proceso 0 usaría `MPI_Send(a,size*size,MPI_INT,i,99,MPI_COMM_WORLD);` y dejando que los receptores usen `MPI_Recv(a,1,xpose,0,99,MPI_COMM_WORLD,&status)` para trasponerla.

```
#include <stdio.h>
#include <mpi.h>
#define size 10

int my_id, nproc, sizeofint, a[size][size], b[size], i, j;
MPI_Datatype diag;
MPI_Status status;

int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  /* Crea tipo para la diagonal */
  MPI_Type_vector(size, 1, size+1, MPI_INT, &diag);
  MPI_Type_commit(&diag);
  if (my_id == 0) {
    for (i=0; i<size; i++)
      for (j=0; j<size; j++) a[i][j] = i*10+j;
    for (i=1; i<nproc; i++)
      MPI_Send(a,1,diag,i,99,MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(b,size,MPI_INT,0,99,MPI_COMM_WORLD,&status);
    printf("Yo soy: %d\n",my_id);
    for (i=0; i<size; i++) printf("%3d ",b[i]);
```

```
    printf("\n");
  }
  MPI_Finalize();
}
```

Figura 25. Definiendo un tipo derivado para la diagonal de una matriz

MPI_Type_indexed permite especificar un diseño de datos no continuos donde el desplazamiento entre bloques sucesivos no necesariamente es el mismo. Esto permite recolectar entradas arbitrarias de un arreglo y enviarlas en un mensaje, o recibir un mensaje y difundir las entradas recibidas dentro de posiciones arbitrarias de un arreglo. *MPI_Type_hindexed* es análogo a *MPI_Type_indexed* pero con los desplazamientos dados en *bytes*.

oldtype 

count = 3, blocklength = (2,3,1) stride = (0,3,8)

newtype 

En la Figura 27 se ilustra como se puede definir una matriz triangular superior y como hacer para enviarla a otro proceso. Para poder utilizar un tipo de dato que haya sido definido es necesario que *MPI* lo reconozca; es decir, una vez definido debe usarse una rutina de *cometido*, *MPI_Type_commit*, para poder utilizar el nuevo tipo en cualquier rutina de envío o recepción.

```
#include <stdio.h>
#include <mpi.h>
#define size 100

int my_id, nproc, sizeofint, a[size][size], i, j;
MPI_Datatype col, xpose;
MPI_Status status;

int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  /* Obtiene la extension de los enteros */
  MPI_Type_extent(MPI_INT, &sizeofint);
  /* Crea tipo para una columna */
  MPI_Type_vector(size, 1, size, MPI_INT, &col);
  /* Crea tipo para la traspuesta */
  MPI_Type_hvector(size, 1, sizeofint, col, &xpose);
  MPI_Type_commit(&xpose);
  if (my_id == 0) {
    for (i=0; i<size; I++)          /* Llena la matriz */
      for (j=0; j<size; j++) a[i][j] = i*10+j;
```

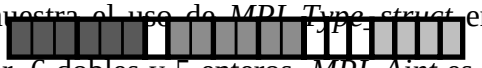
```
    for (i=1; i<nproc; I++)          /* Envia la traspuesta */
      MPI_Send(a,1,xpose,i,99,MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(a,size*size,MPI_INT,0,99,MPI_COMM_WORLD,&status);
    printf("Yo soy: %d\n",my_id);
    for (i=0; i<size; i++)
      for (j=0; j<size; j++) {
       if ((i*10+j)%10 == 0) printf("\n");
       printf("%3d ",a[i][j]);
      }
    printf("\n");
  }
  MPI_Finalize();
}
```

Figura 26. Definiendo un tipo derivado para la matriz traspuesta

MPI_Type_struct define el tipo de dato derivado más general, puesto que es una modificación del Hindexado donde cada bloque puede contener diferentes tipos de datos. La intención es permitir la descripción de arreglos de estructuras como un solo tipo.

oldtypes 

count = 3, blocklength = (2,3,4) stride = (0,7,16)

newtype 

La Figura 28 muestra el uso de *MPI_Type_struct* en donde se define una estruactura que contiene un caracter, 6 dobles y 5 enteros. *MPI_Aint* es un tipo predefinido de *MPI* (MPI array of int) para el arreglo de desplazamientos. Se esta suponiendo que los dobles estan alineados en dos palabras.

```
#include <stdio.h>
#include <mpi.h>
#define size 10
#define half size*(size + 1)/2

int my_id, nproc, sizeofint, a[size][size], b[half],
    disp[size], blocklen[size], i, j;
MPI_Datatype upper;
MPI_Status status;

int main (int argc, char** argv)
{
MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  /* Calcula el comienzo y tamano de cada fila */
  for (i=0; i<size; i++) {
    disp[i] = size*i + i;
    blocklen[i] = size - i;
  }
```

```
  /* Crea tipo para la triangular superior */
  MPI_Type_indexed(size, blocklen, disp, MPI_INT, &upper);
  MPI_Type_commit(&upper);
  if (my_id == 0) {
    for (i=0; i<size; i++)
      for (j=0; j<size; j++) a[i][j] = i*10+j;
    for (i=1; i<nproc; i++)
      MPI_Send(a,1,upper,i,99,MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(b,half,MPI_INT,0,99,MPI_COMM_WORLD,&status);
    printf("Yo soy: %d\n%",my_id);
    for (i=0; i<half; i++) printf("%3d ",b[i]);
    printf("\n");
  }
  MPI_Finalize();
}
```

Figura 27. Código para definir y enviar la triangular superior

4.2. EMPAQUETAMIENTO DE DATOS

Con el objetivo de dar mayor flexibilidad y de ser compatible con las ideas de otras bibliotecas como *PVM* y *Parmacs, MPI* provee mecanismos para empaquetar datos no continuos para que sean enviados como un *buffer* continuo y para esparcirlos nuevamente en localidades discontinuas en el proceso receptor. Además, las operaciones de *packed* y *unpacked* dan a *MPI* capacidades que de otra manera no podrían lograrse como el poder recibir un mensaje en partes, donde la operación de recepción dependa del contenido de las partes que forman el mensaje. También, estas operaciones facilitan el desarrollo de bibliotecas adicionales de comunicación basadas sobre *MPI*.

```
MPI_Datatype my_type;
MPI_Datatype type[3] = {MPI_CHAR, MPI_DOUBLE, MPI_INT};
int          blocklen[3] = {1, 6, 5};
MPI_Aint     dsip[3] = {0, sizeof(double), 7*sizeof(double)};

MPI_Type_struct(3, blocklen, disp, type, &my_type);
MPI_Type_commit(&my_type);
MPI_Send(a, 1, my_type, dest, tag, MPI_COMM_WORLD);
```

Figura 28. Ilustrando *MPI_Type_struct*

El manejo de *packed* y *unpaked* en *MPI* se logra con las siguientes tres rutinas que estan resumidas en la Figura 29 y 30.

MPI_Pack permite copiar en un *buffer* (*outbuf*) continuo de memoria datos almacenados en

el *buffer* (*inbuf*) de entrada. Cada llamada a la rutina *MPI_Pack* modificará el valor del parámetro *position* incrementándolo según el valor de *incount*.

MPI_Unpack permite copiar en un (o varios) *buffer* (*outbuf*) lo datos almacenados en el *buffer* continuo (*inbuf*) de entrada. Cada llamada a la rutina *MPI_Unpack* modificará el valor del parámetro *position* incrementándolo según el valor de *outcount*.

MPI_Pack_size permite encontrar cuanto espacio se necesitara para empaquetar un mensaje y, así, manejar la solicitud de espacio de memoria para el *buffer*.

En las Figura 31 se ilustra el uso de estas funciones. El proceso con rango 0 empaqueta los datos de cierto experimento: etiqueta del experimento, número de prueba y un conjunto de medidas (e.g. temperaturas), y los envía al proceso 1 quien realiza ciertos cálculos con ellos.

En C se puede usar las estructuras *union*, que permite solapar sobre la misma dirección de memoria distintas estructuras de datos, para evitarse el empaquetamiento y desempaquetamiento de datos. El programa que sigue (Figura 32) muestra como se puede hacer. Lo que se quiere es pasar una cadena de 20 caracteres, un entero y un real del proceso 0 a los otros. Para esto se solapa una cadena de 28 caracteres (se asume que los enteros y los reales son de 4 *bytes* cada uno) sobre esta estructura (la *union*). Se trabaja normalmente sobre la estructura, pero al momento de enviarla o recibirla se usa la cadena de caracteres solapada sobre ella; desafortunadamente en Fortran no hay algo similar.

MPI_Pack(inbuf, incnt, datatype, outbuf, outsize, position, comm)

Empaqueta mensajes

E	inbuf	*buffer* de entrada
E	incnt	número de componentes de entrada
E	datatype	tipo de dato de cada componente de entrada
S	outbuf	*buffer* de salida
E	outsize	tamaño del *buffer* de salida en *bytes*
ES	position	posición actual en el *buffer*
E	comm	comunicador para mensajes empaquetados

```
int MPI_Pack(void *inbuf, int incnt, MPI_Datatype datatype, void *outbuf, int outsize, int *position,
             MPI_Comm comm)
MPI_PACK(INBUF, INCNT, DATATYPE, OUTBUF, OUTSIZE, POSITION, COMM, IERROR)
   <tipo> INBUF(*), OUTBUF(*)
    INTEGER INCNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR
```

MPI_Unpack(inbuf, insize, position, outbuf, outcnt, datatype, comm)

Desempaqueta mensajes

E	inbuf	*buffer* de entrada
E	insize	tamaño del *buffer* de entrada en *bytes*
ES	position	posición actual en el *buffer* en bytes
S	outbuf	*buffer* de salida
E	outcnt	número de componentes a ser desempaquetados
E	datatype	tipo de dato de cada componente de salida
E	comm	comunicador para mensajes empaquetados

int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf, int outcnt, MPI_Datatype datatype, MPI_Comm comm)

MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF, OUTCNT, DATATYPE, COMM, IERROR)
<tipo> INBUF(*), OUTBUF(*)
INTEGER INSIZE, POSITION, OUTCNT, DATATYPE, COMM, IERROR

Figura 29. *MPI_Pack y MPI_Unpack*

MPI_Pack_size(incnt, datatype, comm, size)

Obtener el tamaño de `incnt` *datos del tipo* `datatype`

E	incnt	argumento *count* de *MPI_Pack*
E	datatype	argumento *datatype* de *MPI_Pack*
E	comm	argumento *comm* de *MPI_Pack*
S	size	limite superior para el tamaño del mensaje empaquetado en bytes

int MPI_Pack_size(int incnt, MPI_Datatype datatype, MPI_Commm comm, int *size)

MPI_PACK_SIZE(INCNT, DATATYPE, COMM, SIZE, IERROR)
INTEGER INCNT, DATATYPE, COMM, SIZE, IERROR

Figura 30. *MPI_Pack_size*.

```
#include <stdio.h>
#include <mpi.h>
#define maximot 20

main(int argc,char** argv)
{
   int my_id,i,j,tag,position,numexp,numval,longi,tam;
   float temper[maximot];
   char nombrexp[30];
   int nproc;                 /* Numero de procesadores       */
   char *buffer;
   char nhost[20];
   MPI_Status estado;

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

```
   MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
   MPI_Comm_size(MPI_COMM_WORLD,&nproc);

   tag=25;
   position=0;
   if (my_id==0) {
     printf(" Nombre del experimento: \n");
     gets(nombrexp);
     printf(" Numero del experimento: \n");
     scanf("%d",&numexp);
     printf(" Cuantos valores son (menos de 20):\n ");
     scanf("%d",&numval);
     for (i=0;i<numval;i++) {
       printf("valor %d:\n",i );
       scanf("%f",&temper[i]);
     }
     longi = strlen(nombrexp)+1;
     tam = (sizeof(char)*longi) + 3*sizeof(int) + numval* (sizeof(float));
     MPI_Send(&tam,1,MPI_INT,1,tag,MPI_COMM_WORLD);
     buffer = (char *)malloc(tam);
     MPI_Pack(&longi,1,MPI_INT,buffer,tam,&position,MPI_COMM_WORLD);
     MPI_Pack(nombrexp,strlen(nombrexp)+1,MPI_CHAR,buffer,tam,
              &position,MPI_COMM_WORLD);
     MPI_Pack(&numexp,1,MPI_INT,buffer,tam,&position,MPI_COMM_WORLD);

     MPI_Pack(&numval,1,MPI_INT,buffer,tam,&position,MPI_COMM_WORLD);
     MPI_Pack(temper,numval,MPI_FLOAT,buffer,tam,&position,MPI_COMM_WORLD);
     MPI_Send(buffer,position,MPI_PACKED,1,tag,MPI_COMM_WORLD);
   }
   else {
     MPI_Recv(&tam,1,MPI_INT,0,tag,MPI_COMM_WORLD,&estado);
     buffer = (char *)malloc(tam);
     MPI_Recv(buffer,tam,MPI_PACKED,0,tag,MPI_COMM_WORLD,&estado);
     MPI_Unpack(buffer,tam,&position,&longi,1,MPI_INT,MPI_COMM_WORLD);
     MPI_Unpack(buffer,tam,&position,nombrexp,longi,MPI_CHAR,MPI_COMM_WORLD);
     MPI_Unpack(buffer,tam,&position,&numexp,1,MPI_INT,MPI_COMM_WORLD);
     MPI_Unpack(buffer,tam,&position,&numval,1,MPI_INT,MPI_COMM_WORLD);
     MPI_Unpack(buffer,tam,&position,temper,numval,MPI_FLOAT,MPI_COMM_WORLD);
/*   ************************************************************
     Aqui el proceso 1 (UNO) puede hacer cualquier operacion
     sobre los datos recibidos
    ************************************************************/
    gethostname(nhost,20);
    printf("Yo soy la maquina %s\n",nhost);
     printf(" Nombre del experimento: %s\n",nombrexp);
     printf(" Numero del experimento: %d\n",numexp);
     printf(" Numero del valores: %d\n",numval);
     for (i=0;i<numval;i++) {
       printf("valor %d: %.2f\n",i,temper[i]);
     }
   }
   MPI_Finalize();
   }
```

Figura 31. Ilustración del uso de *MPI_Pack* y *MPI_Unpack*

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

```
#include <mpi.h>
#define structsize 28
struct rec_ {
  char str[20];
  int a;
  float b;
};
union U {
  struct rec_ rec;
  char msg[structsize];
};
union U u;
MPI_Status status;
int my_id, nproc, i, tag = 99, source;
int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  if (my_id == 0) {
    printf("Hello World I'm # %d\n",my_id);
    strcpy(u.rec.str,"Hola Mundo  ");

    for (i=1;i<nproc; i ++) {
      rec.rec.a = 20+i;
      rec.rec.b = 1.67+i;
      MPI_Send(u.msg,structsize,MPI_CHAR,i,tag,MPI_COMM_WORLD);
    }
  }
  else {
    MPI_Recv(rec.msg,structsize,MPI_CHAR,source,98,MPI_COMM_WORLD,&status);
    printf("Soy:%d y recibi mensaje de:%d   str:%s   a:%d   b: %5.2f\n",
           my_id,source,u.rec.str,u.rec.a,u.rec.b);
    }
  MPI_Finalize();
}
```

Figura 32. Usando *uniones* en C para evitar empaquetar datos

5. PROCESAMIENTO POR GRUPOS

Todas las comunicaciones en *MPI* están relacionas a un comunicador *communicator* el cual contiene un contexto y un grupo. El grupo denota a un conjunto de procesos. El procesamiento por grupos permite a *MPI* cubrir una serie de debilidades presentes en algunas bibliotecas de pases de mensajes e incrementa sus capacidades de portabilidad, eficiencia y seguridad. El procesamiento por grupos agrega elementos como: división de procesos, transmisión de mensajes sin conflictos, extensibilidad para los usuarios (especialmente los que escriben bibliotecas de funciones) y seguridad.

Un grupo, en *MPI*, es un conjunto ordenado de procesos; donde cada proceso es identificado

a través de un n*úmero* entero denominado rango (*rank*). El grupo especial *MPI_GROUP_EMPTY* sirve para denotar aquel grupo que **NO** tiene miembros. Mientras, la constante *MPI_GROUP_NULL* es un valor usado para referirse a un grupo no válido.

Un elemento fundamental en el intercambio de información en *MPI* es el *comunicador* (*communicator*), el cual es un objeto opaco con un número de atributos y reglas que gobiernan su creación, uso y destrucción. Tal *comunicador* especifica un dominio de comunicación, por ejemplo *MPI_COMM_WORLD*. Un *comunicador* puede ser usado para las comunicaciones entre los procesos de un mismo grupo, en cuyo caso se habla de un *intracomunicador* (*intracommunicator*), o para comunicaciones punto a punto (las únicas permitidas) entre procesos de diferentes grupos los cuales se denominan *intercomunicadores* (*intercommunicator*).

5.1. CREANDO COMUNICADORES

Como se mencionó anteriormente, las etiquetas de los mensajes (*tag*) proveen un mecanismo para distinguir entre mensajes con distintos propósitos. Sin embargo, no son suficientes para diseños modulares. Por ejemplo, consideremos una aplicación que llama una rutina de una librería. Es importante que las etiquetas usadas en la librería sean diferentes de las usadas en el resto de la aplicación, de lo contrario podría haber confusión de mensajes. Sin embargo, no hay forma de que el usuario sepa cuales etiquetas son usadas por la librería ya que muchas veces son generadas dinámicamente. Los *comunicadores* proveen una solución a este problema a continuación se muestran algunas funciones que permiten usar los comunicadores en forma más flexible.

5.1.1. Creando comunicadores a partir de otros comunicadores

En la Figuras 33 y 34 se muestran algunas funciones que pueden ser usadas para crear (y eliminar) comunicadores a partir de otros comunicadores y se explican a continuación.

MPI_Comm_dup permite crear un nuevo comunicador (*newcomm*) compuesto de los mismos procesos que hay en *comm* pero con un nuevo contexto para asegurar que comunicaciones efectuadas para distintos propósitos no sean confundidas.

MPI_Comm_dup(comm, newcomm)
Crea un nuevo comunicador: mismo grupo, nuevo contexto

E	comm	communicador
S	newcom	comunicador

int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
INTEGER COMM, NEWCOMM, IERROR

Figura 33. ***MPI_Comm_dup.***

MPI_Comm_split puede crear un nuevo comunicador que sólo incluya un subconjunto de un grupo de procesos dado. Estos procesos después pueden comunicarse entre ellos sin temor de conflictos con otras operaciones concurrentes. *MPI_Comm_split* es colectiva, implicando que todos los procesos asociados con *comm* deben ejecutarla. Se crea un nuevo comunicador (*newcomm*) por cada valor distinto de *color* y diferente de *MPI_UNDEFINED*. Si un proceso llama *MPI_Comm_split* con *color = MPI_UNDEFINED*, entonces no formara parte del nuevo comunicador y en este se devolverá *newcomm = MPI_COMM_NULL*.

MPI_Comm_split(comm, color, key, newcomm)

Particiona un grupo en subgrupos disjuntos

E	comm	communicador
E	color	control para los subgrupos
E	key	control para los rangos de los procesos
S	newcom	comunicador

MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR

MPI_Intercomm_create(comm, leader, bridge_comm, rleader, tag, newintercomm)

Crea un intercomunicador

E	comm	intracommunicador local
E	leader	rango del líder local según comm
E	bridge_comm	communicador de enlace o puente (MPI_COMM_WORLD es suficiente en la mayoría de los casos)
E	rleader	rango del líder remoto según bridge_comm
E	tag	etiqueta de seguridad
S	newintercomm	intercomunicador nuevo

MPI_Intercomm_create(MPI_Comm comm, int leader, MPI_Comm bridge_comm, int rleader, int tag,
MPI_Comm *newintercomm)

```
MPI_INTERCOMM_CREATE(COMM, LEADER, BRIDGE_COMM, RLEADER, TAG, NEWINTERCOMM,
                     IERROR)
  INTEGER COMM, LEADER, BRIDGE_COMM, RLEADER, TAG, NEWINTERCOMM,  IERROR
```

MPI_Comm_free(comm)

Destruye un comunicador

ES	comm	communicador

```
MPI_Comm_free(MPI_Comm *comm)
MPI_COMM_FREE(COMM, IERROR)
  INTEGER COMM, IERROR
```

Figura 34. ***MPI_Comm_split, MPI_Intercomm_create*** **y** ***MPI_Comm_free***

Los procesos asociados al nuevo comunicador tendrán rangos que comienzan de cero, con el orden determinado por el valor de *key*. En caso de empates, se identificarán de acuerdo a su orden en el comunicador viejo.

MPI_Intercomm_create construye un intercomunicador para enlazar los procesos de dos grupos. La llamada es colectiva para todos los procesos en la unión de los dos grupos. Los procesos deben llamar la función con el respectivo comunicador local (*comm*) y argumento *leader* idéntico respecto a cada uno de los dos grupos. Ambos líderes deben coincidir en el argumento *bridge_comm* y deben proveer en *rleader* el rango del otro líder respecto al dominio de *bridge_comm*. Ambos valores de *tag* deben ser iguales. *MPI_Comm_free* permite destruir un comunicador creado con alguna de la funciones anteriores.

El programa en la Figura 35 ilustra estas funciones. Primero se usa *MPI_Comm_split* para dividir los procesos en dos grupos. Todos aquellos que tienen rango par en *MPI_COMM_WORLD* van a un grupo y los de rango impar a otro grupo. El rango en el nuevo grupo (*new_id*) sigue el orden del rango en el grupo original (*my_id*). *MPI_Intercomm_create* se llama para enlazar los dos grupos creados. Los procesos 0 dentro de cada grupo serán los líderes, que corresponden al proceso 0 y 1 dentro del grupo original. Una vez creado el intercomunicador, cada proceso del segundo grupo le envía un mensaje al proceso respectivo del primer grupo (0 del segundo a 0 del primero, ..., *n*-1 del segundo a *n*-1 del primero). Se asume que el número de procesos es par. Finalmente los nuevos comunicadores son eliminados.

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

```
#include <mpi.h>
int my_id, new_id, i;
MPI_Status status;
MPI_Comm newcomm, intercomm;
int main (int argc, char** argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

  MPI_Comm_split(MPI_COMM_WORLD, my_id%2, my_id, &newcomm);
  MPI_Comm_rank(newcomm, &new_id);

  if (my_id%2 == 0) {
    MPI_Intercomm_create(newcomm, 0, MPI_COMM_WORLD, 1, 99, &intercomm);
    MPI_Recv(&i, 1, MPI_INT, MPI_ANY_SOURCE, 0, intercomm, &status);
    printf("my_id: %d  new_id: %d  recibio mensaje: %d  de: %d\n",
          my_id, new_id, i, status.MPI_SOURCE);
  }

  else {
    MPI_Intercomm_create(newcomm, 0, MPI_COMM_WORLD, 0, 99, &intercomm);
    MPI_Send(&new_id, 1, MPI_INT, new_id, 0, intercomm);
  }
  MPI_Comm_free(intercom);
  MPI_Comm_free(newcomm);
  MPI_Finalize();
}
```

Figura 35. Ilustrando el uso de *MPI_Comm_split* y *MPI_Intercomm_create*

5.1.2. Creando grupos y comunicadores a partir de grupos

MPI_Comm_group(comm, group)

Obtiene el grupo asociado a un comm

E	comm	communicador
S	group	grupo asociado a *comm*

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

MPI_COMM_GROUP(COMM, GROUP, IERROR)
INTEGER COMM, GROUP, IERROR

MPI_Group_union(group1, group2, newgroup)

Crea un nuevo grupo a partir de la unión de dos grupos

E	group1	primer grupo
E	group2	segundo grupo
S	newgrup	todos los elementos del primer grupo seguidos de los elementos del segundo grupo

int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_Group_intersection(group1, group2, newgroup)

Crea un nuevo grupo a partir de la intersección de dos grupos

E	group1	primer grupo
E	group2	segundo grupo
S	newgrup	todos los elementos del primer grupo que tam,bien estan en el segundo grupo

int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

Figura 36. ***MPI_Comm_group, MPI_Group_union,y MPI_Group_intersection***

MPI_Group_difference(group1, group2, newgroup)

Crea un nuevo grupo a partir de la diferencia de dos grupos

E	group1	primer grupo
E	group2	segundo grupo
S	newgrup	todos los elementos del primer grupo que no estan en el segundo grupo

int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

MPI_Group_free(group)

Elimina o destruye un grupo

ES	group	grupo a eliminar

int MPI_Group_free(MPI_Group *group)

MPI_GROUP_FREE(GROUP,IERROR)
INTEGER GROUP, IERROR

MPI_Group_incl(group, n, ranks, newgroup)

Crea un nuevo grupo que consiste de n elementos de group

E	group	grupo original
E	n	número de elementos en el arreglo *ranks* y tamaño del nuevo grupo
E	ranks	arreglo con los rangos de los procesos en *group* que estarán en *newgroup*
S	newgroup	nuevo grupo con orden definido por *ranks*

int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI_Group_excl(group, n, ranks, newgroup)

Crea un nuevo grupo que se obtiene eliminando n elementos de group

E	group	grupo original
E	n	número de elementos en el arreglo *ranks*
E	ranks	arreglo con los rangos de los procesos en *group* que NO estarán en *newgroup*
S	newgroup	nuevo grupo con orden definido por *group*

int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

Figura 37. ***MPI_Group_difference, MPI_Group_free, MPI_Group_incl y MPI_Group_excl.***

MPI_Group_range_incl(group, n, ranges, newgroup)

Crea un nuevo grupo a partir de elementos de group

E	group	grupo original
E	n	número de ternas en el arreglo *ranges*
E	ranges	arreglo de ternas de enteros de la forma (primer rango, último rango, desplazamiento) indicando grupos de procesos en *group* que estarán en *newgroup*
S	newgroup	nuevo grupo con orden definido por *ranges*

int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)

MPI_GROUP__RANGE_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI_Group_range_excl(group, n, ranges, newgroup)

Crea un nuevo grupo excluyendo elementos de group

E	group	grupo original
E	n	número de ternas en el arreglo *ranges*
E	ranges	arreglo de ternas de enteros de la forma (primer rango, último rango, desplazamiento) indicando grupos de procesos en *group* que NO estarán en *newgroup*
S	newgrup	nuevo grupo con orden definido por *group*

int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)

MPI_GROUP__RANGE_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
INTEGER GROUP, N, RANGES(3,*), NEWGROUP, IERROR

MPI_Comm_create(comm, group, newcomm)

Crea un nuevo intracomunicador para los elementos en group

E	comm	communicador

E	group	el grupo que es un subconjunto del grupo asociado a *comm*
S	newcomm	nuevo comunicador

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)
MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
    INTEGER COMM, GROUP, NEWCOMM, IERROR
```

Figura 38. ***MPI_Group_range_incl, MPI_Group_range_excl y MPI_Comm_create***

En esta sección se describen algunas operaciones para la manipulación de grupos. Estas operaciones son locales y no requieren de comunicación entre procesos. El grupo base sobre el cual todos los otros grupos son definidos, es el grupo asociado con *MPI_COMM_WORLD* que se puede obtener con la funcion *MPI_COMM_GROUP*. En C el tipo para los grupos es *MPI_Group*. En las Figuras 36,37 y 38 se muestran varias funciones que operan sobre grupos y permiten crear comunicadores en base a ellos.

Para *MPI_Group_incl*, si *group* es {a,b,c,d,e} y *ranks* = (3,0,2) entonces *newgroup* es {d,a,c}. Para *MPI_Group_range_incl*, si *group* es {a,b,c,d,e,f,g,h,i,j} y *ranges* es ((6,7,1),(1,6,2),(0,9,4)), entonces la primera terna especifica los procesos {g,h}, la segunda {b,d,f} y la tercera {a,e,i}. La llamada crea el grupo {g,h,b,d,f,a,e,i}. Cada rango calculado debe ser válido y todos los rangos deben ser distintos, de lo contrario hay un error. *MPI_Group_excl* y *MPI_Group_range_excl* funcionan en forma análoga.

6. COMUNICACION SIN BLOQUEO

Las rutinas *MPI_Send* y *MPI_Recv* son con bloqueo; esto significa que una llamada a send es bloqueada hasta tanto el *buffer* de envío pueda ser reusado (podríamos decir que el mensaje haya sido enviado). Similarmente, la función de recepción se bloqua hasta tanto el *buffer* de recepción contenga el mensaje esperado.

MPI_Isend(buf, count, datatype, dest, tag, comm, request)

Envio de mensajes sin bloqueo

E	buff	dirección del *buffer* de envio
E	count	número de elementos a enviar
E	datatype	tipo de datos de los elementos del *buffer* de envio
E	dest	rango del proceso destino
E	tag	etiqueta del mensaje

E	comm	comunicador
S	request	objeto para determinar si send ha sido completado

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
              MPI_Request *request)
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,IERROR)
    <tipo> BUF(*)
    INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,IERROR
```

Figura 39. *MPI_Isend*.

El rendimiento en muchos sistemas puede mejorarse si se logra solapar las comunicaciones y el cálculo, sobre todo en aquellos sistemas que poseen un controlador inteligente de las comunicaciones. El *multihebrado* es un mecanismo para lograr tal solapamiento. Mientras una *hebra* del programa permanece bloqueada esperando por la finalización de la comunicación, otra puede estar ejecutándose en el mismo procesador. Un mecanismo alterno, que usualmente da mejor rendimiento, es el uso de comunicación sin bloqueo.

Una rutina de solicitud de envío lo inicia pero no lo completa. Debido a que la llamada retorna antes de que el *buffer* de envío sea reusable, es necesario utilizar una llamada de completación para verificar que tal copia ha sido realizada. Con un *hardware* apropiado, la transferencia de datos de la memoria del enviador puede realizarse de manera concurrente con los cálculos hechos después de la inicialización del envío y antes de que este se complete. La llamada a una rutina de recepción sin bloqueo funciona de manera similar.

Las rutinas que provee *MPI* para la transferencia sin bloqueo tiene una semántica similar a las rutinas con bloqueo, excepto por el uso del prefijo *I* (*Inmediato*) y que a *MPI_Isend* se le agrega el parametro *status* que posteriormente permitira determinar si el envío ha sido completado (Figuras 39 y 40).

MPI_Irecv(buf, count, datatype, source, tag, comm, request)

Recepción de mensajes sin bloqueo

S	buff	dirección del *buffer* de recepción
E	count	número de elementos a recibir
E	datatype	tipo de datos de los elementos del *buffer* de recepción
E	source	rango del proceso enviador o *MPI_ANY_SOURCE*
E	tag	etiqueta del mensaje o *MPI_ANY_TAG*
E	comm	comunicador

S	request	objeto para determinar si recv ha sido completado

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

MPI_ISEND(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,IERROR)
<tipo> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,IERROR

Figura 40. *MPI_Irecv*.

Las funciones *MPI_Wait* y *MPI_Test* son usadas para completar el envío y la recepción sin bloqueo. Una permite esperar por la finalización de una operación y la otra para verificar si ha sido completada (Figura 41). Con *request* se especifica la operación asociada (*MPI_Isend* o *MPI_Irecv*) y en *status* se devuelve información sobre la operación completada.

Con *MPI_Iprobe* y *MPI_Probe* (Figura 42) se puede chequear si se han recibido mensajes de una fuente específica (*source*) y con una etiqueta (*tag*) determinada, sin efectivamente recibir el mensaje. Tal como con *send* y *recv*, la primera, *MPI_Iprobe*, es sin bloqueo y retorna *flag = true* en caso de haber un mensaje que se ajuste a los parámetros definidos, mientras que la segunda, *MPI_Probe*, se bloquea hasta que llegue un mensaje con las características indicadas. Una vez que se sabe que llego el mensaje, se puede recibir usando *MPI_Recv*. Con estas funciones es posible que dos procesos intercambien información sin previamente saber la longitud de la misma. Una vez que *MPI_Iprobe* o *MPI_Probe* hayan sido exitosas, se puede revisar el objeto *status* con *MPI_Get_count* (Figura 42) para obtener la información respectiva y usarla en *MPI_recv*.

MPI_Wait(request, status)
Espera a que la operación asociadacon request *ha sido completada*

ES	request	asa al *request*
S	status	objeto status

int MPI_Wait(MPI_Request *request, MPI_Status *status)

MPI_WAIT(REQUEST, STATUS, IERROR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

MPI_Test(request, flag, status)
Permite determinar si la operacion asociada con request *ha sido completada*

ES	request	asa al *request*

S	flag	cierto si la operación ha sido completada
S	status	objeto status

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

Figura 41. Funciones para chequear o esperar por un envio o recepción

MPI_Iprobe(source, tag, comm, flag, status)

Revisa si han llegado mensajes sin realmente recibirlos - Sin bloqueo

E	source	rango del enviador
E	tag	etiqueta del mensaje
E	comm	comunicador
S	flag	cierto si hay un mensaje con el respectivo *source* y *tag*
S	status	objeto status

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)

MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_Probe(source, tag, comm, status)

Revisa si han llegado mensajes sin realmente recibirlos - Con bloqueo

E	source	rango del enviador
E	tag	etiqueta del mensaje
E	comm	comunicador
S	status	objeto status

int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

MPI_Get_count(staus, datatype, count)

Calcula el número de elemntos del tipo datatype *recibidos*

E	status	*status* devuelto por *MPI_Recv*, *MPI_Iprobe*, etc.
E	datatype	tipo de dato de los elementos recibidos
S	count	número de elementos recibidos

```
int MPI_Get_count(int staus, MPI_Datatype datatype, int *count)
MPI_GET_COUNT(STAUS, DATATYPE, COUNT, IERROR)
    INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR)
```

Figura 42. *MPI_Iprobe, MPI_Probe y* ***MPI_Get_count***

7. DEFINIENDO TOPOLOGÍAS SOBRE LOS PROCESOS

Una topología es un atributo extra y opcional que se le puede dar a un intra-comunicador más no a un inter-comunicador. Esta topología provee una forma conveniente de nombrar los procesos de un grupo y podría ayudar en su asignación a los procesadores.

En muchas aplicaciones paralelas un enumeramiento lineal de los procesos 0, 1, 2, ..., *n*-1, no refleja las necesidades de comunicación entre ellos ya que usualmente siguen cierto patrón topológico tal como mallas de dos o tres dimensiones De manera más general, el arreglo lógico de los procesos puede ser descrito a través de un grafo. Este arreglo lógico será referido como *Topología Virtual*. Es necesario distinguir entre la topología virtual de los procesos y la topología física de interconexión de los procesadores. La topología virtual puede ser explotada por el sistema en la asignación de los procesos a los procesadores en caso de que ayude a mejorar el rendimiento en una máquina específica. La descripción de la topología virtual depende sólo de la aplicación y es independiente de la máquina. El mapeo proceso-procesador esta fuera del alcance de *MPI*.

7.1. TOPOLOGIAS VIRTUALES

Los patrones de comunicación entre un grupo de procesos se pueden describir mediante un grafo en donde los vértices representan los procesos y los enlaces conectan los procesos que se comunican. En la mayoría de los casos las comunicaciones son simétricas y los grafos se asumen simétricos; es decir, los enlaces se consideran bidireccionales.

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)

Figura 43. Relación entre el rango y las coordenadas Cartesianas para una malla 3x4 2D.

Los grafos resultan suficientes para todas las aplicaciones, sin embargo, en muchos casos la estructura del grafo es regular y una descripción detallada en términos de grafos resulta inconveniente y posiblemente menos eficiente a la hora de correr. Muchos programas paralelos usan topologías como anillos y mallas, las cuales se definen en forma sencilla especificando el número de dimensiones y los procesos en cada coordenada. Por lo tanto estos casos se consideran en forma particular. Las coordenadas de los procesos en una estructura Cartesiana se enumeran a partir del 0 avanzando primero por las filas, tal como se muestra en la Figura 43 en donde 12 procesos son organizados en una malla bidimensional 3x4. La flexibilidad de *MPI* para definir topologías incluye la posibilidad de solapar diferentes topologías Cartesianas. La información de la topología esta asociada con comunicadores.

7.2 CREACION Y MANEJO DE TOPOLOGIAS CARTESIANAS

7.2.1. Creación de topologías Cartesianas

Para definir topologías Cartesianas de dimensión arbitraria, *MPI* provee la función *MPI_Cart_create*. Para cada coordenada se especifica si la estructura es periódica o no. En el caso 1D, es lineal si no es periódica y un anillo en caso contrario. Para el caso 2D, puede ser un rectángulo, cilindro o torus al pasar de no periódico a periódico en una dimensión a completamente periódico (Figura 44).

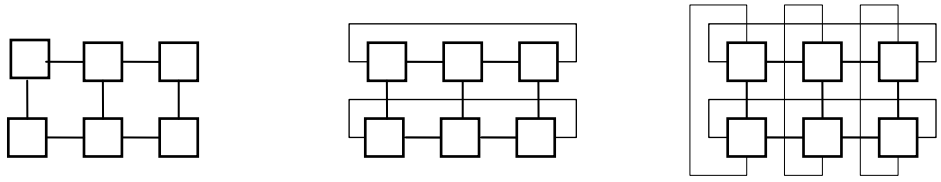


Figura 44. Caso 2D: rectángulo, cilindro y torus.

MPI_Cart_create (Figura 45) es una función colectiva y devuelve, en cada proceso que la llama, un asa a un nuevo comunicador que incluye la información sobre la topología Cartesiana. En caso de que *reorder = true* el rango de cada proceso en el nuevo grupo coincide con el rango en el viejo grupo, de lo contrario la función puede decidir reordenar los procesos. Si el tamaño

total de la malla es menor que el tamaño del grupo viejo, algunos procesos recibirán MPI_COMM_NULL en analogía con *MPI_Comm_split*.

7.2.2. Función de ayuda para la construcción de topologías cartesianas

MPI_Dims_create (Figura 46) es una función local y ayuda a seleccionar una distribución balanceada de los procesos por coordenada, dependiendo del número de procesos en el grupo y de restricciones adicionales que se especifiquen. Esta función, por ejemplo, se puede usar para particionar un grupo en una topología *n*-dimensional. Los elementos del arreglo *dims* describen una malla Cartesiana con *ndims* dimensiones y un total de *nnodes* nodos. Si se especifican elementos del arreglo *dims* antes de hacer la llamada, se puede restringir la operación. Si *dims[i]* es un número positivo, la rutina no modificara el número de nodos en la dimensión *i*; solo aquellas dimensiones con *dims[i] = 0* son modificadas. En casos de inconsistencias, se retornará un error. El resultado de *MPI_Dims_create* puede ser usado como entrada en *MPI_Cart_create*. En la Tabla 4 se dan ejemplos de varias llamadas a esta función.

MPI_Cart_create(comm_old, ndims, dims, periods, reorder, comm_cart)

Describe estructuras cartesianas de dimensión arbitraria.

E	comm_old	comunicador de entrada
E	ndims	n*úmero* de dimensiones de la malla cartesiana
E	dims	arreglo de enteros de tamaño *ndims* que contiene el n*úmero* de procesadores en cada dimensión
E	periods	arreglo de valores lógicos que indican si la malla es periódica en cada dimensión; es decir, si el último proceso esta conectado con el primero
E	reorder	valor lógico que indica si los procesos serán o no reordenados
S	comm_cart	comunicador con la nueva topología cartesiana

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int dims, int *periods, int reorder,
                    MPI_Comm comm_cart)
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    LOGICAL PERIODS(*), REORDER
    INTEGER COMM_OLD, NDIMS, DIMS, COMM_CART, IERROR
```

Figura 45. ***MPI_Cart_create***

dims antes	llamada	dims después	observación
(0,0)	MPI_Dims_create(6,2,dims)	(3,2)	3x2=6

(0,0)	MPI_Dims_create(7,2,dims)	(7,1)	7x1=7
(0,3,0)	MPI_Dims_create(6,3,dims)	(2,3,1)	2x3x1=6
(0,3,0)	MPI_Dims_create(7,3,dims)	error	No hay forma de dividir adecuadamente 7 nodos en 3 dimensiones

Tabla 4. Ejemplos de llamadas a MPI_Dims_create

MPI_Dims_create(nnodes, ndims, dims)

Dado el rango devuelve las coordenadas

E	nnodes	número de nodos en la malla
E	ndims	número de dimensiones Cartesianas
S	coords	arreglo de enteros que contiene las coordenadas cartesianas del respectivo proceso

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)
MPI_CART_COORDS(NNODES, NDIMS, DIMS, IERROR)
    INTEGER NNODES, NDIMS, DIMS(*), IERROR
```

Figura 46. *MPI_Dims_create*.

7.2.3. Funciones de consultas para topologías Cartesianas

Una vez que se ha definido una topología cartesiana, puede ser necesario hacer consultas sobre su estructura. Para esto *MPI* provee las funciones que se muestran en la Figuras 47 y 48.

MPI_Cartdim_get retorna el número de dimensiones de la topología Cartesiana asociada con *comm*. La llamada a esta función usando el comunicador asociado con la Figura 43 retornaría *ndims = 2*. *MPI_Cart_get* retorna información sobre la topología Cartesiana asociada con *comm*. *maxdims* debe ser por lo menos *ndims* retornado por *MPI_Cartdim_get*. Con respecto a la Figura 43, con *dims = (3,4)* en *coords* se devolverán las coordenadas del proceso que llama la función, por ejemplo, para el proceso con rango 9, *coords = (2,1).*

MPI_Cart_rank dadas las coordenadas Cartesianas de un proceso, retorna su respectivo rango. Para la Figura 45, para *coords = (2,1)* devuelve *rank = 9*. *MPI_Cart_coords* es el inverso de *MPI_Cart_rank*; dado el rango devuelve las coordenadas. Para la Figura 45, si *rank = 9* la función retornaria *coords = (2,1). maxdims* debe ser por lo menos de tamaño *ndims*.

7.3. CREACION Y MANEJO DE GRAFOS

MPI_Graph_create es una función colectiva que devuelve un nuevo comunicador que lleva consigo la información del grafo (Figura 50). Si *reorder = false*, entonces el rango de cada proceso en el grafo es el mismo que en el grupo viejo, en caso contrario la función podría reordenar los procesos. Si el número de nodos *nnodes* que se especifica al llamar la función es menor que el número de nodos en el grupo de *comm_old*, entonces algunos procesos recibirán *MPI_COMM_NULL* como retorno de la llamada. La llamada será errónea si se trata de definir un grafo con más procesos que los que hay en *comm_old*.

MPI_Cartdim_get(comm, ndims)

Retorna el número de dimensiones asociados con la estructura cartesiana

E	comm	comunicador con estructura cartesiana
S	ndims	n*úmero* de dimensiones de la malla cartesiana

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
    INTEGER COMM, NDIMS, IERROR
```

MPI_Cart_get(comm, maxdims, dims, periods, coords)

Retorna información sobre la estructura cartesiana asociada a comm

E	comm	comunicador con estructura cartesiana
E	maxdims	longitud del arreglo *dims*, *periods* y *coords*
S	dims	número de procesos por cada dimensión cartesiana
S	periods	periodicidad (cierto/falso) de cada dimensión cartesiana
S	coords	arreglo de enteros que contiene las coordenadas cartesianas del respectivo proceso

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int *coords)
MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
    LOGICAL PERIODS(*)
    INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
```

MPI_Cart_rank(comm, coords, rank)

Dadas las coordenadas devuelve el rango

E	comm	comunicador con estructura Cartesiana
E	coords	arreglo de enteros que especifica las coodenadas Cartesianas de un proceso
S	rank	rango del proceso esoecificado

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
MPI_CART_COORDS(COMM, COORDS, RANK, IERROR)
    INTEGER COMM, COORDS(*), RANK, IERROR
```

Figura 47. *MPI_Cartdim_get, MPI_Cart_get, y MPI_Cart_rank*

MPI_Cart_coords(comm, rank, maxdims, coords)

Dado el rango devuelve las coordenadas

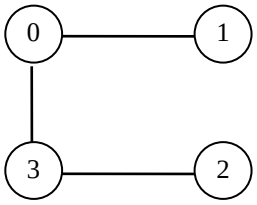
E	comm	comunicador con estructura cartesiana
E	rank	rango de un proceso en el grupo de *comm*
E	maxdims	longitud del arreglo *coords*
S	coords	arreglo de enteros que contiene las coordenadas cartesianas del respectivo proceso

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
    INTEGER COMM, RANK, MAXDIMS, COORDS, IERROR
```

Figura 48. *MPI_Cart_coords*

Proceso	Vecinos
0	1, 3
1	0
2	3
3	0, 2




```
nnodes = 4
index = (2, 3,  4, 6)
edges = (1, 3, 0, 3, 0, 2)
```

Figura 49. Ejemplo de la definición de un grafo .

MPI_Graph_create(comm_old, nnodes, index, edges, reorder, comm_graph)

Retorna los rangos de los procesos vecinos

E	comm_old	comunicador de entrada
E	nnodes	n*úmero* de nodos en el grafo. los nodos son numerados del 0 a *nnodes-1*
E	index	arreglo que contiene en la posición *i* el n*úmero* total de vecinos de los primeros *i* nodos
S	edges	arreglo que contiene la lista de vecinos de los nodos 0, 1,...*nnodes-1* almacenados consecutivamente
S	reorder	valor lógico que indica si los rangos de los procesos serán o no reordenados
S	comm_graph	comunicador con la nueva topología de grafo

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges, int reorder,
                     MPI_Comm comm_graph)
```

```
MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES, REORDER, COMM_GRAPH,
                 IERROR)
    LOGICAL REORDER
    INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*), COMM_GRAPH, IERROR
```

Figura 50. ***MPI_Graph_create***

nnodes representa el número de nodos en el grafo y van números 0, 1, ..., *nnodes*-1. El *i*-esimo elemento del arreglo *index* indica cuanto vecinos tienen los primeros *i* nodos del grafo y el tamaño de *index* debe ser *nnodes*. edges es una representación plana de los enlaces y su tamaño debe ser igual al número de enlaces que tiene el grafo que se esta definiendo. En la Figura 49 se da un ejemplo de lo que serían los respectivos parámetros para el grafo de 4 nodos mostrado en la misma Figura.

MPI_Graphdims_get(comm, nnodes, nedges)

Retorna el número de nodos y enlaces de del grafo asociado con comm

E	comm	comunicador de entrada
S	nnodes	número de nodos en el grafo
S	nedges	número de enlaces en el grafo

```
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
    INTEGER COMM, NNODES, NEDGES, IERROR
```

MPI_Graph_get(comm, maxindex, maxedges, index, edges)

Retorna index *y* edges *tal como fueron proporcionados en* MPI_Graph_cerate

E	comm	comunicador de entrada
E	maxindex	longitud del vector *index* en el proceso que llama la función
E	maxedges	longitud del vector *edges* en el proceso que llama la función
S	index	arreglo de enteros que describen el número de vecinos de los nodos
S	edges	arreglo de enteros que describen los vecinos de los nodos

```
int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index, int *edges)
MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES, IERROR)
    INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR
```

Figura 51. ***MPI_Graphdims_get*** **y** ***MPI_Graph_get***

MPI_Graphdims_get retorna el número de nodos y de enlaces del grafo asociado con el comunicador *comm*. El número de nodos es idéntico al tamaño del grupo asociado con *comm* y

nedeges puede ser usado para proveer el tamaño correcto de *index* y *edges* en *MPI_Graph_get*. Para el ejemplo de la Figura 49 se devolvería *nnodes = 4* y *nedges = 6* (recuérdese que los enlaces son bidireccionales y en realidad cada enlace mostrado en el gráfico de la Figura 79 equivale por dos enlaces).

MPI_Graph_neighbors_get(comm, rank, nneighbors)

Retorna el número de vecinos que tiene el proceso con rango rank

E	comm	comunicador de entrada
E	rank	rango del proceso
S	nneighbors	número de vecinos del proceso con rango *rank*

```
int MPI_Graph_heighbors_get(MPI_Comm comm, int rank, int *nneighbors)
MPI_GRAPH_NEIGHBORS_GET(COMM, RANK, NNEIGHBORS, IERROR)
    INTEGER COMM, RANK, NNEIGHBORS, IERROR
```

MPI_Graph_neighbors(comm, rank, maxneighbors, neighbors)

Retorna los vecinos que tiene el proceso con rango rank

E	comm	comunicador de entrada
E	rank	rango del proceso
E	maxneighbors	tamaño del arreglo *neighbors* (*nneighbors* devuelto por *MPI_Graph_neighbors_get*)
S	neighbors	arreglo de rangos que especifica los vecinos del proceso con rango *rank*

```
int MPI_Graph_heighbors(MPI_Comm comm, int rank, int maxneighbors, int *neighbors)
MPI_GRAPH_NEIGHBORS (COMM, RANK, MAXNEIGHBORS, NEIGHBORS, IERROR)
    INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*), IERROR
```

Figura 52. *MPI_Graph_neighbors_get y MPI_Graph_neighbors.*

MPI_Graph_get devuelve los arreglos *index* y *edges* tal como fueron dados en *MPI_Graph_create*. *maxindex* y *maxedges* deben ser al menos de longitud *nnodes* y *nedeges*, respectivamente, en concordancia con lo que retorno *MPI_Graphdims_get*.

MPI_Graph_neighbors_count devuelve el número de vecinos para el proceso con rango *rank*. Este valor puede ser usado para suplir el valor correcto para el tamaño del arreglo *neighbors* en *MPI_Graph_neighbors*. *MPI_Graph_neighbors* devuelve la parte del vector *edges* asociada con el proceso con rango *rank*. Para el ejemplo de la Figura 49, si *rank = 3* devolvería *neighbors = 0, 2*.

Finalmente, un proceso puede recibir un comunicador que tenga asociada una topología no

conocida previamente. *MPI* provee la función local *MPI_Topo_test* (Figura 53) para resolver este tipo de incertidumbre. Dependiendo del caso, *status* puede tomar alguno de los tres valores que se indican en la Tabla 5.

Valor de *status*	**Tipo de topología asociada**
MPI_GRAPH	grafo
MPI_CART	topología Cartesiana
MPI_UNDEFINED	sin topología asociada

Tabla 5. Valores posibles de *status* que puede devolver *MPI_Topo_test*

MPI_Topo_test(comm, status)

Retorna el tipo de topología asociada con comm

E	comm	comunicador de entrada
S	status	tipo de topoplogía asociada a *comm*

```
int MPI_Topo_test(MPI_Comm comm, int *status)

MPI_TOPO_TEST(COMM, STATUS, IERROR)
    INTEGER COMM, STATUS, IERROR
```

Figura 53. *MPI_Topo_test*

BIBLIOGRAFIA

1) Foster, I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley, New York, 1995. http://www.mcs.anl.gov/dbpp
2) Quinn, M. *Parallel Computing: Theory and Practice*, McGraw-Hill, New York, 1994.
3) Gropp, W., Lusk, E., y Skjellum, A. *MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Massachusetts, 1994.
4) Snir, M., Dongarra, J., y otors. *MPI: The Complete Reference.* MIT Press, Massachusetts, 1996.

SITIOS DE INTERES EN INTERNET

1) *CeCalCULA*:

   http://www.cecalc.ula.ve/

2) Información sobre *MPI: Message Passing Interfae*:

   http://www.mcs.an.gov/mpi/index.html

3) Información actualizada sobre versiones de *MPI*:

   http://www.mcs.anl.gov/Projects/mpi/implementations.html

   http://www.mpi.nd.edu/MPI

   http://www.erc.msstate.edu/mpi/implementations.html

4) High Performance Computing and Communications Glossary:

   http://nhse.npac.syr.edu/hpccgloss/hpccgloss.html

5) 3Com Glossary of Networking Terms se puede accesar desde:

   http://www.3com.com/