

TECNOLOGIAS DE LA INFORMACION Y COMUNICACIONES

SISTEMAS OPERATIVOS II

Curso Introducción a MPI

PRESENTA:

Adrián Prado Medina

Sídney Ricardo García Rodríguez

Luis Fernando Duron Castañeda

Juan Ignacio Ornelas Carreón

DOCENTE:

Eduardo Flores Gallegos

PABELLON DE ARTEAGA AGS, 24 DE NOVIEMBRE 2017

Contenido

[1. CURSO MPI 5](#_Toc499587510)

[1.1 Objetivo del curso 6](#_Toc499587511)

[1.2 Audiencia objetivo 6](#_Toc499587512)

[1.3 Requisitos previos 6](#_Toc499587513)

[1.4 Fundamentos de Computación Paralela 6](#_Toc499587514)

[1.4.1 Introducción 6](#_Toc499587515)

[1.4.2 Objetivos 7](#_Toc499587516)

[1.4.3 Diseño de programa paralelo 13](#_Toc499587517)

[1.4.4 Descomposición del problema 14](#_Toc499587518)

[1.4.5 Equilibrio de carga 15](#_Toc499587519)

[1.4.6 Tiempo de ejecución 16](#_Toc499587520)

[1.4.7 Tiempo de inactividad del proceso 16](#_Toc499587521)

[2. Empezando con MPI 17](#_Toc499587522)

[2.1 Primeros pasos con MPI 17](#_Toc499587523)

[2.1.1 Introducción 17](#_Toc499587524)

[2.2 Objetivos 17](#_Toc499587525)

[2.2.1 El estándar MPI 18](#_Toc499587526)

[2.2.2 Metas de MPI 18](#_Toc499587527)

[2.2.3 Tipos de rutinas de MPI 18](#_Toc499587528)

[2.2.4 Comunicaciones y mensajes punto a punto 19](#_Toc499587529)

[2.2.5 Modos de comunicación y criterios de finalización 19](#_Toc499587530)

[2.2.6 Compilación y ejecución de programas MPI 21](#_Toc499587531)

[2.3 Un primer programa: ProcessColors 21](#_Toc499587532)

[3. Estructura del Programa MPI 28](#_Toc499587533)

[3.1 Estructura del programa 28](#_Toc499587534)

[3.2 Archivos de encabezado 28](#_Toc499587535)

[3.3 Convenciones de nomenclatura 29](#_Toc499587536)

[3.4 Rutinas MPI y valores de entorno 29](#_Toc499587537)

[3.5 MPI Handles 30](#_Toc499587538)

[3.5.1 Tipos de datos MPI 30](#_Toc499587539)

[3.5.2 Tipos de datos MPI básicos 31](#_Toc499587540)

[3.5.3 Tipos de datos MPI especiales 32](#_Toc499587541)

[3.6 Inicializando MPI 33](#_Toc499587542)

[3.6.1 Comunicadores 33](#_Toc499587543)

[3.6.2 Obtención de información del comunicador: Clasificación 34](#_Toc499587544)

[3.6.3 Obtener la Información del comunicador: tamaño 35](#_Toc499587545)

[3.7 MPI de terminación 36](#_Toc499587546)

[3.8 ¡Hola Mundo! mk.2 36](#_Toc499587547)

[3.8.1 Salida del programa de ejemplo 37](#_Toc499587548)

[4. Comunicación Punto A Punto 38](#_Toc499587549)

[4.1 Comunicación punto a punto 38](#_Toc499587550)

[4.2 Origen y destino 38](#_Toc499587551)

[4.3 Mensajes 39](#_Toc499587552)

[4.4 Envío y recepción de mensajes 39](#_Toc499587553)

[4.5 Bloqueo de envío y recepción 40](#_Toc499587554)

[4.5.1 Bloqueo de envío y recepción 40](#_Toc499587555)

[4.5.2 Envío de un mensaje: MPI\_SEND 40](#_Toc499587556)

[4.5.3 Recibir un mensaje: MPI\_RECV 40](#_Toc499587557)

[4.5.4 Ejemplo: Enviar y recibir 42](#_Toc499587558)

[4.5.5 Recepción de comodines 42](#_Toc499587559)

[4.5.6 Tamaño del mensaje 43](#_Toc499587560)

[4.5.7 Ejemplo de programa comodín 44](#_Toc499587561)

[4.5.8 Comportamiento en tiempo de ejecución 44](#_Toc499587562)

[4.5.9 Bloqueo y finalización 44](#_Toc499587563)

[4.6 Envíos y recepciones sin bloqueo 45](#_Toc499587564)

[4.6.1 Envíos y recepciones sin bloqueo 45](#_Toc499587565)

[4.6.2 Posiciones de publicación, finalización y solicitud 45](#_Toc499587566)

[4.6.3 Publicación de envíos sin bloqueo 45](#_Toc499587567)

[4.6.4 Publicación recibe sin bloquear 46](#_Toc499587568)

[4.6.5 Finalización: espera y prueba 47](#_Toc499587569)

[4.6.6 Ventajas y desventajas 49](#_Toc499587570)

[4.6.7 Ejemplo de envío / recepción 49](#_Toc499587571)

[4.7 Modos de envío y recepción 50](#_Toc499587572)

[4.8 Ejercicio 4 - Juego de la vida 50](#_Toc499587573)

[4.9 Ejercicio 4: búsqueda paralela 50](#_Toc499587574)

[4.10 Autocomprobación - Comunicaciones punto a punto 51](#_Toc499587575)

[5. Comunicación de datos no contiguos o tipos de datos mixtos 54](#_Toc499587576)

[5.1 Comunicación de datos no contiguos o tipos de datos mixtos 54](#_Toc499587577)

[5.1.1 Introducción 54](#_Toc499587578)

[5.1.2 Objetivos 54](#_Toc499587579)

[5.2 Envío de mensajes múltiples 55](#_Toc499587580)

[5.3 Almacenamiento en búfer 57](#_Toc499587581)

[5.3.1 Almacenamiento en búfer 57](#_Toc499587582)

[5.3.2 Una manera incorrecta y tentadora de ampliar el almacenamiento en búfer 58](#_Toc499587583)

[5.3.3 Almacenamiento en búfer de la manera correcta: empacar y desempaquetar 58](#_Toc499587584)

[5.4 Tipos de datos derivados 59](#_Toc499587585)

[5.4.1 Tipos de datos derivados 59](#_Toc499587586)

[5.4.2 Tipos de datos definidos por el usuario 60](#_Toc499587587)

[5.4.3 Construir tipos de datos derivados 62](#_Toc499587588)

[5.4.4 Coincidencia de mensajes y desajuste 63](#_Toc499587589)

[5.4.5 Combinación de tipos de datos derivados 64](#_Toc499587590)

[5.4.6 Obtención de información sobre sus tipos derivados 67](#_Toc499587591)

[5.5 Ejercicio 5 - Juego de la vida 67](#_Toc499587592)

[5.6 Ejercicio 5: búsqueda paralela 72](#_Toc499587593)

[5.6.1 Ejercicio 73](#_Toc499587594)

[6. Comunicaciones colectivas 74](#_Toc499587595)

[6.1 Comunicaciones colectivas 74](#_Toc499587596)

[6.1.1 Objetivos 74](#_Toc499587597)

[6.2 Sincronización de barreras 75](#_Toc499587598)

[6.3 Transmisión 75](#_Toc499587599)

[6.4 Reducción 77](#_Toc499587600)

[6.5 Reunir 78](#_Toc499587601)

[6.6 Dispersión 79](#_Toc499587602)

[6.7 Operaciones avanzadas 79](#_Toc499587603)

[6.8 Ejercicio 6 - Juego de la vida 80](#_Toc499587604)

[6.9 Ejercicio 6: búsqueda paralela 80](#_Toc499587605)

[6.10 Autocomprobación - Comunicaciones colectivas 81](#_Toc499587606)

# CURSO MPI

La interfaz de paso de mensajes (MPI). Es una biblioteca estándar de subrutinas (Fortran) o conocidas como funciones (C) en el que implementa un programa de paso de mensajes. Permite la coordinación de un programa y que se ejecute como procesos múltiples en un entorno de memoria distribuida, flexible como para ser usado en un sistema de memoria compartida. Esta biblioteca MPI se estandariza de una manera muy poderosa, ya que los programas deben compilarse y ejecutarse en cualquier plataforma. Implementa arquitecturas, ofreciendo funcionalidades, tipos de comunicación y rutinas especiales en operaciones colectivas comunes, además de manejar topologías y datos definidos por el usuario.

## Objetivo del curso

Permitir el desarrollo de programas de transferencia de mensajes en Fortran o C.

## Audiencia objetivo

Programadores e investigadores interesados ​​en usar o escribir programas paralelos para resolver problemas complejos.

## Requisitos previos

La experiencia puede ser nula en MPI o programación paralela para realizar este curso. Sin embargo, es necesario comprender la programación de la computadora. Si no está familiarizado con los conceptos de programación paralela, le sugerimos que tome el tutorial Explicación de la computación paralela antes de tomar este.

## Fundamentos de Computación Paralela

### Introducción

La complejidad de los problemas de la ciencia computacional en áreas de investigación como biomoléculas, clima y modelado climatológico continúan en crecimiento. El poder computacional de recursos informáticos no ha seguido el ritmo de crecimiento y por eso se han desarrollado métodos informáticos paralelos. En la computación paralela muchas computadoras con múltiples procesadores están interconectadas para que los procesadores individuales puedan trabajar al mismo tiempo y resolver una parte pequeña del problema complejo que es mucho mayor para solo una computadora. Para aprovechar esto, un programa debe escribirse en paralelo para que la ejecución del programa ocurra más de una ocasión (proceso), este proceso representa un subprograma ejecutándose de manera automática en un procesador físico. Su objetivo principal en este tipo de programación es aumentar el rendimiento computacional relacionado con el rendimiento por la ejecución en serie del mismo programa.

Se debe combinar las capacidades de hardware y software para una programación paralela, estos son:

* La mutua conexión entre procesadores y memoria tiene una comunicación rápida entre los procesos individuales, y la transferencia rápida de datos, hacia y desde la memoria.
* Un protocolo debe estar disponible para la intercomunicación entre procesos.
* La separación de los algoritmos computacionales debe ser posible, así como los datos de entrada en subproblemas individuales.
* Debe haber un mecanismo disponible para asignar subproblemas individuales a procesos separados.

### Objetivos

En esta lección, aprenderás sobre:

* Diseño básico de hardware de computación paralela.
* Modelos de programación paralela.
* Diseño de programa paralelo.

#### Diseño Básico de Hardware para Computación Paralela

En 1972, Se propuso una clasificación de arquitecturas informáticas que ahora se conoce como taxonomía1 ​​de Flynn, hecha por Michael Flynn. Se basa en la cantidad de instrucciones y flujos de datos disponibles en las arquitecturas. Puede haber secuencias de instrucciones simples o múltiples, y haber flujos de datos únicos o múltiples.

Las cuatro clasificaciones son:

* Instrucción única de datos individuales (SISD)
* Datos únicos de instrucción múltiple (MISD)
* Datos Múltiples Instrucciones Simples (SIMD)
* Múltiples datos múltiples de instrucción (MIMD)}

En esta lección, discutiremos las dos arquitecturas más adecuadas para computación paralela, SIMD y MIMD.

##### Arquitectura SIMD

Una arquitectura de datos múltiples de instrucción única (SIMD) consta de un solo ordenador que se usa para el control; Y un conjunto de unidades computacionales separadas (chips de microprocesador), cada uno con su propia memoria.

Su principal desventaja es que de las unidades computacionales permanecen deshabilitadas durante largo tiempo si un programa contiene grandes secciones de código, ya que su ejecución depende de una o más sentencias condicionales.

##### Arquitecturas MIMD

A diferencia de SIMD, en esta arquitectura cada procesador (MIMD) Múltiples Datos de Instrucción Múltiple tiene una unidad de control como una unidad computacional. Cada procesador ejecuta su propio subprograma independiente de los demás procesadores dentro del sistema, son arquitecturas asincrónicas.

Hay dos tipos principales de arquitectura MIMD:

* Memoria compartida
* Memoria distribuida.

**Memoria compartida MIMD**

La arquitectura MIMD de memoria compartida es una colección de procesadores y unidades de memoria interconectadas por algún tipo de red, que puede estar basada en bus o switch. En la mayor parte de las arquitecturas de memoria compartida, el número de procesadores está limitado (entre 2 y 16). Por esto la cantidad de datos que se procesa está limitada debido al ancho de banda de la interconexión de memoria entre los procesadores.

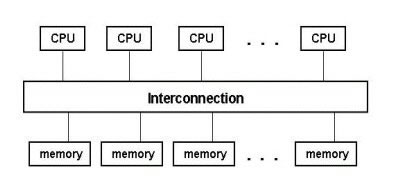


Figura 1.1: Arquitectura MIMD de memoria compartida

Fuera de su memoria principal las computadoras tienen dos tipos de memorias adicionales de llamadas: registro y caché. La memoria de registro representa una memoria interna que contiene datos sobre los que el procesador trabaja, generalmente solo está disponible en cantidades limitadas.

La memoria caché representa una memoria intermedia que es más lenta que la memoria de registro, se puede acceder mucho más rápido que la memoria principal. El acceso a la memoria caché es importante porque está basada en las arquitecturas de memoria compartida en bus.

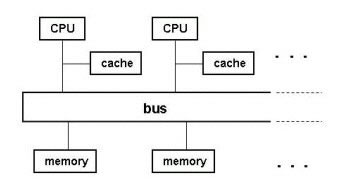


Figura 1.2: Arquitectura MIMD de memoria compartida basada en bus (uso de caché en arquitecturas de memoria compartida basadas en bus).

**MIMD de memoria distribuida**

En esta arquitectura cada procesador tiene su propia memoria privada.

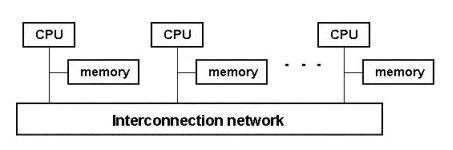


Figura 1.3: arquitectura MIMD de memoria distribuida

Una computadora con esta arquitectura de memoria distribuida es un conjunto de computadoras seriales llamadas nodos. Trabajan juntas para resolver un problema de computo, cada nodo tiene su acceso rápido a su propia memoria local y el acceso a la memoria de otros nodos a través de una red de interconexión.

**Clústeres SMP**

Una de las arquitecturas de la informática paralela es el procesamiento múltiple simétrico (SMP). Esta es una combinación de arquitecturas MIMD de memoria compartida y memoria distribuida. Es un clúster de nodos de procesamiento de memoria compartida conectados mediante una red de interconexión. En un clúster SMP, cada procesador dentro de un nodo tiene acceso a una memoria compartida. Como es una arquitectura de memoria distribuida, desde que se proporciona un procesador dentro de un nodo, su capacidad de acceder a datos, dentro de otros nodos requiere técnicas de programación paralelas. Un ejemplo de SMP es el simulador de tierra japonés.

#### Modelos de programación en paralelo

Hay dos enfoques principales para la programación paralela. Son:

* Modelo De Paso De Mensajes
* Modelo Paralelo De Datos Basado En Directivas

Estos enfoques se diferencian por la forma en que se organiza el espacio de direcciones de la computadora paralela.

##### Modelo de paso de mensajes

En el modelo de paso de mensajes, cada proceso envía y recibe mensajes para comunicarse con otros procesos. Es el modelo más utilizado para la programación paralela en arquitecturas de memoria distribuida. La interfaz de paso de mensajes (MPI) es la interfaz de programación de aplicaciones (API) estándar para el envío de mensajes.

##### Modelo paralelo de datos basados ​​en directivas

En este modelo los lenguajes de programación hacen que el código de serie sea paralelo desde que se agrega una directiva como comentarios en el código. Estas directivas dicen cómo distribuir datos y trabajar en los procesadores. Muestra detalles de cómo hacer la distribución de datos, cálculo y comunicaciones se dejan al compilador. Este modelo se usa en arquitecturas de memoria compartida, ya que el espacio de memoria disponible facilita la escritura de compiladores.

Una API estándar que utiliza este modelo para la programación paralela de memoria compartida en arquitecturas de memoria compartida es OpenMP.

### Diseño de programa paralelo

Los programas paralelos se basan en varias partes de un programa en serie y este se comunica mediante llamadas a la biblioteca, y se pueden dividir en cuatro clases:

* **Llamadas utilizadas para inicializar, administrar y finalizar las comunicaciones.**

Se usan para iniciar las comunicaciones, identificar el número de procesos que se utilizan, crear subgrupos de procesadores y que proceso está ejecutando un programa.

* **Llamadas utilizadas para comunicarse entre pares de procesos.**

También llamadas operaciones de comunicaciones punto a punto, se dividen en diferentes tipos de operaciones de envío y recepción.

* **Llamadas que realizan operaciones de comunicación entre grupos de procesos.**

Son las operaciones que proporcionan la sincronización, los tipos de operaciones de comunicación definidas entre procesos y que realizan operaciones de comunicación / cálculo.

* **Llamadas utilizadas para crear tipos de datos arbitrarios.**

Estos son flexibles al tratar con estructuras de datos complicadas.

Su principal objetivo al escribir un programa paralelo es obtener un buen rendimiento que se obtendría con una versión en serie. Se debe considerar varios problemas, como diseñar su código paralelo. Estos son los que descomponen el problema, equilibrio de carga, tiempo que tarda en ejecutar y el tiempo que dura inactivo el proceso.

### Descomposición del problema

Uno de los requisitos para la programación paralela es la manera de separar los algoritmos requeridos y los datos de entrada en pequeños subproblemas a procesadores separados a esto se le denomina descomposición del problema.

Hay dos tipos de descomposición:

* Descomposición de dominio
* Descomposición funcional

El método que debe usar depende de las restricciones de su aplicación.

#### Descomposición del dominio

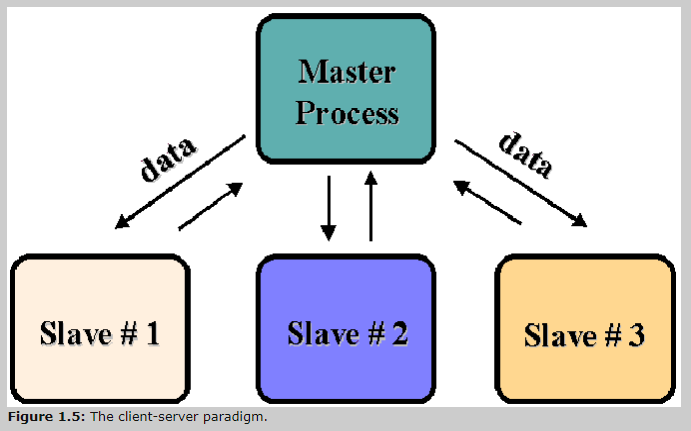
También conocido como paralelismo de datos, estos se dividen en pequeñas piezas que son del mismo tamaño y son asignados a diferentes procesadores. Estos procesadores solo funcionan en la parte de datos que les corresponde, sin dejar de tomar en cuenta que deben comunicarse frecuentemente para el intercambio de datos.

Esta descomposición tiene como ventaja mantener un solo flujo de control. Un algoritmo de datos paralelos se conforma por una secuencia de instrucciones y se inician cuando la instrucción anterior a finalizado. El Programa único de datos múltiples (SPMD) se basa en este modelo donde su código es igual en todos los procesadores.

#### Descomposición funcional

La descomposición funcional o también llamada paralelismo de tarea es una estrategia de descomposición muy eficiente, ya que el problema se descompone en muchas pequeñas tareas (funciones) se asignan a los procesadores conforme se están disponibles, y a los procesadores que terminan pronto su trabajo se les asigna un trabajo adicional.

La descomposición funcional se implementa en un paradigma cliente-servidor, las tareas son asignadas a un grupo de esclavos (procesos) por medio de un proceso que también realiza tareas (proceso maestro). El paradigma cliente-servidor puede adaptarse a cualquier programa. A medida que cada procesador finaliza su tarea, se le asigna una nueva entrada.



### Equilibrio de carga

Se divide el trabajo de igual manera entre todos los procesos disponibles. De esta manera se asegura que los procesos no permanezcan deshabilitados mientras que los demás procesos trabajan en sus subprogramas se asignan y no se desperdicien los recursos de computo. El equilibrio de carga es fácil una vez que los procesos realizan la misma operación. Cuando se tiene variaciones de tiempo grandes es mejor adoptar una estrategia para resolver el problema.

### Tiempo de ejecución

El tiempo de ejecución siempre es la principal preocupación en la programación paralela ya que es esencial para comparar y mejorar todos los programas.

Existen tres componentes componen el tiempo de ejecución:

* Tiempo de cálculo
* Tiempo de inactividad
* Tiempo de comunicación

Es el tiempo dedicado a realizar cálculos en los datos. El tiempo de inactividad es el tiempo que un proceso espera los datos de otros procesadores, durante este tiempo el procesador no tiene tiempo útil. El tiempo de comunicación es el tiempo que tardan los procesos en enviar y recibir mensajes. Su costo se puede medir en latencia y ancho de banda.

### Tiempo de inactividad del proceso

Es importante cuando el tiempo que tardan los procesos disminuyen y así reducir el impacto en el tiempo de ejecución. Existen estrategias para reducirlo como comunicación y calculo superpuesto. El uso adecuado de la comunicación y el cálculo no especifico de datos lo pueden hacer posible. Pero con desventaja de que en la práctica es difícil de interconectar la comunicación con el cálculo, para esto debes ser planificado cómo y cuándo sea posible usar la latencia y escribir su código de una manera adecuada.

# Empezando con MPI

* 1. Primeros pasos con MPI

### Introducción

MPI es una implementación estándar del modelo de paso de mensajes que tiene funciones en *C* o subrutinas en *Fortran* que son insertadas en el código fuente y realiza la comunicación de datos entre los procesos. MPI incluye una o más llamadas a una biblioteca de funciones, queda claro que MPI no es una biblioteca, es una especificación de interfaces y proporcionar un estándar para la escritura de mensajes. Un programa MPI se conforma por dos o más procesos autosuficientes, donde cada uno ejecuta sus códigos que pueden o no ser repetidos en varios procesos, estos procesos se comunican mediante llamadas a rutinas de comunicación MPI y se identifica según su rango.

## Objetivos

El objetivo de esta lección es poder escribir versiones en serie del código, como escribir versiones paralelas. Además de familiarizarse con los siguientes conceptos de la programación de MPI:

* El estándar MPI
* Los objetivos y el alcance de MPI
* Tipos de rutinas MPI
* Comunicaciones y mensajes punto a punto
* Comunicaciones colectivas
* Compilando y ejecutando programas MPI
* Cómo escribir un programa simple de MPI

### El estándar MPI

MPI se desarrolló aproximadamente con ayuda de sesenta personas en representación de cuarenta organizaciones, se definió como MPI en el año de 1994. El cual especifica nombres secuencia de llamadas, resultados en subrutinas y las funciones serian desde *Fortran 77* y *C.* Todos los programas deben compilarse y ejecutarse en toda plataforma que admita MPI. Las implementaciones de MPI-1 están disponibles para un sinfín de plataformas. Además también se ha definido el estándar MPI-2 que proporciona funciones adicionales, como herramientas para los enlaces paralelos de *E / S, C++* y *Fortran 90.*

Algunas de las implementaciones de MPI incluyen partes del estándar MPI-2

### Metas de MPI

MPI proporciona la portabilidad de código fuente, permite implementaciones en una gran variedad de arquitecturas. Ofrece cantidad de funcionalidades, incluyendo series de tipos de comunicación, operaciones colectivas, manejo de topologías y soporte.

Metas fuera del alcance de MPI-1: El mecanismo para lanzar un programa, ya que depende de la plataforma. La gestión de procesos: Cambio en la cantidad de procesos al momento de ejecutarse el código.

### Tipos de rutinas de MPI

El estándar MPI incluye rutinas como:

* Comunicación punto a punto
* Comunicaciones colectivas
* Grupos de procesos
* Topologías de procesos
* Gestión del medio ambiente y consulta

### Comunicaciones y mensajes punto a punto

Es la comunicación directa entre dos procesadores, en el cual uno envía datos y el otro recibe estos. En un envió o recepción de mensaje consiste en un bloque de datos entre procesadores. MPI utiliza 3 elementos que caracterizan el mensaje:

* **Buffer:** Es la ubicación de inicio (son los datos salientes) o los datos entrados (para recibir). El *C* el buffer es la dirección del elemento, en *Fortran* es solo el nombre del elemento donde comienza la transferencia de datos.
* **Tipo de datos:** es el tipo de datos que se enviaran, ejemplo, *Float (Real), int (entero),* En otro tipo de aplicaciones puede ser definido por el usuario
* **Recuento:** Es el número de elementos de tipo de datos que se enviaran.

### Modos de comunicación y criterios de finalización

MPI proporciona flexibilidad para darse a especificar que los mensajes se enviaran, define el procedimiento utilizado para transmitir el mensaje, así como criterios que determinen que se completara la comunicación, ejemplo, un envió se considera completo cuando el se confirma el recibo del mensaje. Cuando se envía un bufer se completa cuando los datos se han copiado en el bufer local, no implica la llegada del mensaje a su destino

Hay cuatro tipos de comunicación disponibles para envíos:

* Estándar
* Sincrónico
* Buffered
* Listo

Para las recepciones existe un modo de comunicación y es cuando los datos que entran han llegado y están disponibles

#### Comunicación de bloqueo y no bloqueo

Un envío de mensajes puede estar bloqueado en la comunicación o no bloqueado, este no regresa de la llamada hasta que la operación se haya completado realmente y garantiza que haya cumplido los criterios de finalización antes de que el proceso de llamada continúe. Un envío o recepción sin bloqueo regresa en el momento sin información sobre si se cumplieron los criterios de finalización. Tiene la ventaja de que el procesador puede hacer otra cosa mientras la comunicación se desarrolla en segundo plano. El procesador puede hacer trabajo útil dejando y probando más tarde si se completó el envío.

#### Grupos de procesos

Es un conjunto ordenado de procesos en la que cada proceso se asocia a un valor que hace referencia como un rango del proceso, también conocido como ID. Sus valores comienzan desde cero y se ejecutan a través de N-1 (N es el número de procesos). Aunque permanezca el mismo número de procesos durante la ejecución del programa.

#### Topologías de proceso

Una topología es un mecanismo que asocia esquemas de identificación con los procesos que pertenecen a un grupo, la topología describe un orden de procesos. MPI admite dos tipos de topologías: careciana (de grilla) y una de gráfico. Estas son virtuales en as que su estructura no puede ser relacionada entre la estructura de proceso. Son usadas para proporcionar eficiente comunicación y la conveniencia de la programación. Según el programa MPI es determinada la topología por el desarrollador de la aplicación.

#### Gestión del entorno y consulta

Algunas rutinas de MPI están disponibles para administrar el estado del medio ambiente, son usadas para varios propósitos cono iniciar y terminar la ejecución de MPI, terminar los procesos pertenecientes a un comunicador determinado, así como el número de procesos y el rango del proceso.

### Compilación y ejecución de programas MPI

El estándar MPI no especifica, y es importante tomar en cuenta las implementaciones que varían de una maquina a otra. Cuando se ejecuta un programa en MPI debe ser vinculado con la biblioteca de MPI, donde se incluirán comandos especiales que se usarán al compilarlo. Se usa un contenedor para ejecutar un código MPI para la implementación u entono por lotes.

## Un primer programa: ProcessColors

El código enumerado a continuación ilustra el uso de comunicaciones punto a punto y colectivas. En este código, una matriz de tres colores (blanco, rojo y verde) se distribuye a todos los procesos creados al comienzo del programa MPI. Cada proceso cambia su color y envía su atributo de color modificado nuevamente al proceso 0, que imprime tanto el color original como el color modificado de cada proceso.

Después de haber definido todas las variables requeridas que se iban a usar en el programa, especificamos nuestro proceso raíz. El proceso raíz se utilizó para transmitir colores a todos los procesos e imprimir los colores de cada uno de nuestros procesos. El proceso 0 fue seleccionado como el proceso raíz.

A continuación, generamos los tres colores separados utilizados para "colorear" nuestros procesos al asignar dinámicamente la memoria para una matriz, array de colores, que contenía ncolor = 3 elementos en los que almacenamos los tres colores: blanco, rojo y verde.

ncolors = 3;

colorArray = (int\*) malloc(sizeof(int) \* ncolors);

for (k = 0; k < ncolors; k++)

{

colorArray[k] = k;

}

Luego comenzamos la codificación MPI primaria y luego accedimos (1) al rango de cada proceso y (2) al número total de procesos creados en tiempo de ejecución.

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &procID);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &nproc);

Utilizamos el comunicador MPI\_COMM\_WORLD, que consta de todos los procesos creados al inicio de la ejecución de MPI.

Luego transmitimos todo el conjunto de colores a cada proceso usando el comando MPI\_BCAST y teníamos cada color de proceso con el color verde (el tercer elemento en colorArray).

MPI\_Bcast(colorArray, ncolors, MPI\_INT, root, MPI\_COMM\_WORLD);

pcolor = colorArray[2];

El rango del procesador que llama a MPI\_BCAST no se especificó porque todos los procesadores en el comunicador MPI\_COMM\_WORLD deben realizar esta llamada de comunicación colectiva. En este caso, fue el proceso raíz el que envió los mismos datos, colorArray, a cada proceso en el comunicador. Además, no se incluyó ningún parámetro de etiqueta en la función MPI\_BCAST. Este parámetro no es necesario porque en esta operación de comunicación colectiva, el número de elementos y el tipo de datos de estos elementos son idénticos en todos los procesos.

A continuación, hicimos que cada proceso que no sea de raíz envíe su color de vuelta al proceso raíz, que luego imprimió el color de cada uno de los procesos nproc. Utilizamos las operaciones de comunicación punto a punto MPI\_SEND y MPI\_RECV para comunicarnos entre los procesos.

Luego cambiamos el color de cada proceso en función de si el rango del proceso era par o impar (incluso rango blanco, rango impar rojo) y si cada proceso no raíz enviaba su color modificado al proceso raíz, que nuevamente imprimía el color de cada uno de los procesos nproc

**Código de muestra**

Aquí proporcionamos el código y la salida para nuestro programa C de muestra, ProcessColors.c.

C:

#include <stdlib.h>

#include <stdio.h>

#include "mpi.h"

main (int argc, char\* argv[])

{

int procID, nproc, root, source, target, tag;

int k, ncolors, pcolor;

int \*colorArray;

char color[10];

MPI\_Status status;

// Set the rank 0 process as the root process

root = 0;

// Generate three colors for color array, where white = 0, red = 1, and green = 2

ncolors = 3;

colorArray = (int\*) malloc(sizeof(int) \* ncolors);

for (k = 0; k < ncolors; k++)

{

colorArray[k] = k;

}

// Initialize MPI

MPI\_Init(&argc, &argv);

// Get process rank

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &procID);

// Get total number of processes specificed at start of run

MPI\_Comm\_size(MPI\_COMM\_WORLD, &nproc);

// Broadcast the array of colors to all processes

MPI\_Bcast(colorArray, ncolors, MPI\_INT, root, MPI\_COMM\_WORLD);

// Color each process 'green' (color = 2)

pcolor = colorArray[2];

// Have each process send its color to the root process

tag = pcolor;

target = 0;

if (procID != root)

{

MPI\_Send(&pcolor, 1, MPI\_INT, target, tag, MPI\_COMM\_WORLD);

}

else

{

for (source = 0; source < nproc; source++)

{

if (source != 0)

{

MPI\_Recv(&pcolor, 1, MPI\_INT, source, tag, MPI\_COMM\_WORLD, &status);

}

switch(pcolor)

{

case 0: sprintf(color, "white");

break;

case 1: sprintf(color, "red");

break;

case 2: sprintf(color, "green");

break;

default: printf("Invalid color\n");

}

printf("proc %d has color %s\n", source, color);

}

printf("\n\n");

}

pcolor = procID%2;

if (pcolor == 0)

{

sprintf(color, "white");

}

else if (pcolor == 1)

{

sprintf(color, "red");

}

else if (pcolor == 2)

{

sprintf(color, "green");

}

// Access new process colors

if (procID != root)

{

MPI\_Send(color, 10, MPI\_CHAR, root, tag, MPI\_COMM\_WORLD);

}

else

{

printf("proc %d has color %s\n", root, color);

for (source = 1; source < nproc; source++)

{

MPI\_Recv(color, 10, MPI\_CHAR, source, tag, MPI\_COMM\_WORLD, &status);

printf("proc %d has color %s\n", source, color);

}

}

free(colorArray);

MPI\_Finalize();

}

Después de compilar el código usando el comando:

mpicc -o ProcessColors ProcessColors.c

…

mpirun –np 8 ProcessColors

...

proc 0 has color green

proc 1 has color green

proc 2 has color green

proc 3 has color green

proc 4 has color green

proc 5 has color green

proc 6 has color green

proc 7 has color green

proc 0 has color white

proc 1 has color red

proc 2 has color white

proc 3 has color red

proc 4 has color white

proc 5 has color red

proc 6 has color white

proc 7 has color red

# Estructura del Programa MPI

## Estructura del programa

Todos los programas MPI cuentan con una misma estructura general la cual es:

include MPI header file

variable declarations

initialize the MPI environment

...do computation and MPI communication calls...

close MPI communications

El archivo de encabezado MPI cuenta con definiciones especificas MPI y prototipos de función. Una vez declaradas las variables, cada uno de los procesos tiende a llamar una rutina MPI la cual inicia con un entorno de transmisión de mensajes.

Después de haber iniciado el entorno MPI, el cálculo puede ocurrir intercalando la comunicación MPI junto con otras rutinas, para interactuar con otros procesos.

Por último, antes de que el programa finalice cada proceso llama una rutina la cual finaliza MPI. No se pueden llamar rutinas MPI después de que se llama la rutina de terminación.

## Archivos de encabezado

Los archivos de cabecera MPI contienen los prototipos para funciones y subrutinas MPI, así como definición de constantes especiales, tipos de datos utilizados por MPI, macros. Para poder realizar esto se debe usar la instrucción “include” junto con el archivo fuente que contiene las llamadas. Las declaraciones se realizan de la siguiente forma:

C:

#include <mpi.h>

## Convenciones de nomenclatura

Los nombres de todas las entidades MPI deberán de comenzar con MPI\_. Esto permitirá distinguir que tienen algo que ver con MPI además de que evitara confusiones entre más variables.

Llamada de una función en C:

MPI\_Init(&argc, &argv)

En el caso de las constantes MPI los nombres deben de estar escritos en mayúscula, por ejemplo, MPI\_COMM\_WORLD, MPI\_REAL, etc. También se cuenta con una lista de constantes, la cual puede ser consultada en internet.

## Rutinas MPI y valores de entorno

Las rutinas MPI son implementadas como funciones en C. Generalmente esta función devolverá un código de salida con el cual se sabrá si la operación de la rutina fue exitosa o no.

En C, la salida de una llamada a una función MPI nos devuelve un “int”.

int err;

...

err = MPI\_Init(&argc, &argv);

if (err == MPI\_SUCCESS) {

...routine ran correctly...

}

...

En cualquier lenguaje, el código obtenido es MPI\_SUCCESS si la rutina no encontró ningún error. Si se llega a producir un error, el entero tendrá un valor el cual indicará el error especifico.

## MPI Handles

MPI mantiene sus propias estructuras de datos internas, las cuales son referenciadas mediante indicadores, los cuales son devueltos de llamadas MPI y pueden llegar a utilizarse en otras llamadas MPI. En C, los identificadores son punteros a tipos de datos creados a través del mecanismo C typedef y las matrices se indexan comenzando en 0. Algunos identificadores de ejemplo son:

* MPI\_SUCCESS: Un entero tanto en C como Fortran, que se usa para probar los códigos de errores.
* MPI\_COMM\_WORLD: En C, es un objeto de tipo MPI\_Comm (comunicador); en Fortran, un numero entero. En cualquiera de los dos lenguajes representa un comunicador predefinido que consta de todos los procesadores.

Estos identificadores, así como muchos más, se definen en el archivo de encabezado.

### Tipos de datos MPI

MPI permite una traducción automática entre las representaciones de un entorno heterogéneo, al poder proporcionar sus tipos de datos de referencia que corresponden a los diversos tipos de datos que son elementales en C y Fortran. Las variables se declaran normalmente, y los nombres MPI son usados como argumentos en las rutinas MPI.

Además, MPI permite la definición de tipos de datos arbitrarios, los cuales son llamados tipos de datos derivados, construidos desde los tipos de datos básicos.

### Tipos de datos MPI básicos

La siguiente tabla muestra los tipos de datos básicos en MPI y su correspondiente tipo en C y Fortran.

|  |  |
| --- | --- |
| Tipo de dato MPI | C |
| MPI\_CHAR | Carácter |
| MPI\_SHORT | Int corto |
| MPI\_INT | Int |
| MPI\_LONG | Long int |
| MPI\_UNSIGNED\_CHAR | Unsigned char |
| MPI\_UNSIGNED\_SHORT | Unsigned short int |
| MPI\_UNSIGNED | Unsigned int |
| MPI\_UNSIGNED\_LONG | Unsigned long int |
| MPI\_FLOAT | Float |
| MPI\_DOUBLE | Doublé |
| MPI\_LONG\_DOUBLE | Long doublé |
| MPI\_BYTE | (nada) |
| MPI\_PACKED–– | (nada) |

Tabla1: Correspondiente a los tipos de datos MPI y los tipos en C

|  |  |
| --- | --- |
| Tipo de dato MPI | Fortran |
| MPI\_INTEGER | Integer |
| MPI\_REAL | Real |
| MPI\_DOUBLE\_PRECISION | Doublé precisión |
| MPI\_COMPLEX | Complex |
| MPI\_CHARACTER | Carácter(1) |
| MPI\_LOGICAL | lógico |
| MPI\_BYTE | (nada) |
| MPI\_PACKED | (nada) |

Tabla2: Correspondiente a los tipos de datos MPI y los tipos en Fortran

### Tipos de datos MPI especiales

En C, podemos encontrar varios tipos de datos especiales (estructuras) que MPI nos proporciona:

* MPI\_COMM: Comunicador
* MPI\_STATUS: Contiene información de estado para llamadas MPI
* MPI\_DATATYPE

Para poder usar este tipo de datos especiales, deben estar en las declaraciones de variables:

MPI\_Comm some\_comm;

Aquí se declara una variable llamada “some\_comm” y que es de tipo MPI\_COMM (comunicador).

## Inicializando MPI

MPI\_INIT es una rutina de inicialización, esta debe de ser la primera en llamarse en cualquier programa MPI. Esta rutina estable el entorno MPI y devolverá un código de error en caso de existir algún problema. Solo debe de ser llamado MPI\_INIT una vez por programa.

C:

int err;

...

err = MPI\_Init(&argc, &argv);

Los argumentos para MPI\_Init son “argc” y “argv”, ya que son las variables que contienen los argumentos de líneas de comandos para el programa

Fortran:

INTEGER IERR

...

CALL MPI\_INIT(IERR)

### Comunicadores

Un comunicador es un identificador en MPI define un grupo de procesos que pueden comunicarse entre sí. Dentro de la lista de argumentos de la llamada MPI se encuentran explícitos los nombres que identifican los mensajes MPI los cuales están incluidos como parámetros.

Puede llegar a existir muchos comunicadores y que un procesador sea parte de un grupo de varios comunicadores diferentes. Cada comunicador tiene enumerado consecutivamente los diferentes procesadores comenzando desde el 0, este número es conocido como rango. También el rango nos permite especificar el origen y el destino de las llamadas de envío y recepción.

MPI proporciona un comunicador básico llamado MPI\_COMM\_WORLD, con el uso de este cada procesador puede comunicarse con cualquier otro procesador.

### Obtención de información del comunicador: Clasificación

Mediante MPI\_COMM\_RANK se puede determinar el rango de un comunicador.

* Los rangos son consecutivos e inician en 0
* Un procesador puede tener diferentes rangos en los diferentes comunicadores a los que pertenece.

Declaración de función C de MPI\_COMM\_RANK

int MPI\_Comm\_rank (MPI\_Comm comm, int \* rank);

El argumento comm es una variable de tipo MPI\_COMM (comunicador). Se puede recurrir a usar MPI\_COMM\_WORLD u otra alternativa seria usar el nombre de otro comunicador el cual este definido en otro lugar. Esta variable se declara:

MPI\_Comm some\_comm;

Hay que tomar en cuenta que el segundo argumento es la dirección del rango que es una variable de tipo entero.

Fortran MPI\_COMM\_RANK

MPI\_COMM\_RANK (COMM, RANK, IERR)

En este caso todos los argumentos (COMM, RANK E IERR) son de tipo entero.

### Obtener la Información del comunicador: tamaño

Un procesador puede determinar un tamaño, es decir, el número de procesadores de cualquier comunicador al que pertenece, esto mediante MPI\_COMM\_SIZE

C MPI\_COMM\_SIZE

int MPI\_Comm\_size(MPI\_Comm comm, int \*size);

El primer argumento es un comunicador de tipo MPI\_COMM. El segundo argumento es la dirección del tamaño de tipo entera.

Fortran MPI\_COMM\_SIZE

MPI\_COMM\_SIZE(COMM, SIZE, IERR)

En este caso todos los argumentos son de tipo entero.

Si el comunicador es MPI\_COMM\_WORLD, el numero de procesadores devueltos por MPI\_COMM\_SIZE será igual al número definido por:

* La entrada de línea de comando a MPIRUN. Por ejemplo  
  % mpirun -np 4 a.out
* Variable de entorno dependiente del sistema (MP\_PROCS). El tamaño que define esta variable es reemplazado por el valor correspondiente que establece MPIRUN, por ejemplo

% setenv MP\_PROCS 4

% a.out

## MPI de terminación

MPI\_FINALIZE es la ultima rutina de MPI que debe de ser llamada en el programa. Finaliza el programa y limpia las estructuras de datos MPI, cancela operaciones que no se terminaron, y así sucesivamente. Este debe de ser llamado por todos los procesos. Una vez llamado MPI\_FINZALIZE ya no se podrá llamar a ninguna otra rutina MPI incluyendo MPI\_INIT.

C:

...

err = MPI\_Finalize();

Fortran:

INTEGER IERR

...

call MPI\_FINALIZE(IERR)

## ¡Hola Mundo! mk.2

El ejemplo que se muestra a continuación, es una modificación del programa tradicional “Hola Mundo”, donde cada procesador imprime su rango y numero total de procesadores en el comunicador MPI\_COMM\_WORLD

C:

#include <stdio.h>

#include <mpi.h>

void main (int argc, char \*argv[]) {

int myrank, size;

MPI\_Init(&argc, &argv); /\* Initialize MPI \*/

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank); /\* Get my rank \*/

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); /\* Get the total number of processors \*/

printf("Processor %d of %d: Hello World!\n", myrank, size);

MPI\_Finalize(); /\* Terminate MPI \*/

}

Hay que tomar en cuenta que este código:

* Utiliza el comunicador predefinido MPI\_COMM\_WORLD.
* No esta probando el estado de error de rutinas.

Fortran:

PROGRAM hello

INCLUDE 'mpif.h'

INTEGER myrank, size, ierr

C Initialize MPI:

call MPI\_INIT(ierr)

C Get my rank:

call MPI\_COMM\_RANK(MPI\_COMM\_WORLD, myrank, ierr)

C Get the total number of processors:

call MPI\_COMM\_SIZE(MPI\_COMM\_WORLD, size, ierr)

PRINT \*, "Processor", myrank, "of", size, ": Hello World!"

C Terminate MPI:

call MPI\_FINALIZE(ierr)

END

Hay que tomar en cuenta que este código:

* Utiliza el comunicador predefinido MPI\_COMM\_WORLD predefinido en mpif.h.
* No está probando el estado de error de rutinas.

### Salida del programa de ejemplo

Al ejecutar el programa en 4 procesadores nos dará como resultado lo siguiente:

Processor 2 of 4: Hello World!

Processor 1 of 4: Hello World!

Processor 3 of 4: Hello World!

Processor 0 of 4: Hello World!

Cada uno de los procesadores ejecuta el mismo código, así como la búsqueda de su rango y el tamaño de impresión. Cada vez que se ejecuta el código, el orden de las líneas de salida puede cambiar.

# Comunicación Punto A Punto

## Comunicación punto a punto

Es la función de comunicación fundame9ntal proceso envía un mensaje y otro lo recibe. Sin embargo, es menos simple en la práctica y hay varios problemas que debe tener en cuenta. ental proporcionada por la biblioteca de MPI.

Las operaciones de comunicación subyacentes son las mismas en ambos casos, pero la interfaz de programación es muy diferente.

Algunos de los objetivos son:

* Origen y destino.
* Mensajes
* Envío y recepción de mensajes
* Bloqueando enviar y recibir
* Envío y recepción sin bloqueo
* Enviar modos

Origen y destino, mensajes y mensajes de envío y recepción) son fundamentales para la comunicación punto a punto en MPI.

## Origen y destino

Un proceso (la fuente) envía y otro proceso (el destino) recibe. Los procesos de origen y destino operan de forma asíncrona. Incluso el envío y la recepción de un solo mensaje normalmente no están sincronizados.

## Mensajes

El mensaje consta de dos partes que son el sobre y el cuerpo del mensaje. Un mensaje de MPI es análogo al sobre de papel alrededor de una carta enviada por correo a la oficina de correos.

Constan de dos partes:

* El sobre.
* El cuerpo del mensaje.

El sobre de un mensaje tiene cuatro partes:

* Fuente.
* Destino.
* Communicator.
* Etiqueta.

El cuerpo del mensaje tiene tres partes:

* Buffer.
* Tipo de datos.
* Recuento.

## Envío y recepción de mensajes

La fuente se determina implícitamente, pero el resto del mensaje se da explícitamente mediante el proceso de envío. Los procesos suelen tener uno o más mensajes que se han enviado pero que todavía no se han recibido, Para recibir un mensaje, unos procesos especifican un sobre de mensaje que MPI compara con los sobres de los mensajes pendientes.

Para recibir un mensaje, un proceso especifica un sobre de mensaje que MPI compara con los sobres de los mensajes pendientes.

## Bloqueo de envío y recepción

### Bloqueo de envío y recepción

Dos de las rutinas básicas de comunicación punto a punto son MPI\_SEND y MPI\_RECV, ya que ambas rutinas bloquean el proceso de llamada hasta que se completa la operación de comunicación.

### Envío de un mensaje: MPI\_SEND

El cuerpo del mensaje contiene los datos que se enviarán: contar elementos de tipo tipo de datos. El sobre del mensaje dice dónde enviarlo. Además, se devuelve un código de error. La sintaxis para MPI\_SEND en C

Todos los argumentos son argumentos de entrada. La función devuelve un código de error.

El argumento de entrada BUF es una matriz; su tipo declarado debe coincidir con el tipo de datos MPI correspondiente dado en DTYPE . ( BUF también podría ser solo una variable escalar con un tipo correcto. En este caso, el argumento COUNT se establecería en 1).

Los argumentos de entrada COUNT , DTYPE , DEST , TAG , COMM son del tipo INTEGER.

El argumento de salida IERR es de tipo INTEGER; contiene un código de error cuando MPI\_SEND regresa.

### Recibir un mensaje: MPI\_RECV

[MPI\_RECV](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1118&w=MPI_RECV#term) toma un conjunto de argumentos similar a MPI\_SEND, pero varios de los argumentos se usan de una manera diferente Los argumentos de origen, etiqueta y comunicador deben coincidir con los de un mensaje pendiente para que se reciba el mensaje.

Si no se utilizan comodines, la llamada solo puede aceptar mensajes del proceso de envío especificado y solo con el valor de etiqueta especificado. Los comodines de Communicator no están disponibles. Si no se utilizan comodines, la llamada solo puede aceptar mensajes del proceso de envío especificado y solo con el valor de etiqueta especificado. Los comodines de Communicator no están disponibles.

En general, el emisor y el receptor deben estar de acuerdo con el tipo de datos del mensaje, y es su responsabilidad garantizar ese acuerdo. Si el emisor y el receptor utilizan tipos de datos de mensaje incompatibles, los resultados no están definidos.

La sintaxis para MPI\_RECV en C y Fortran se muestra a continuación.

C: int MPI\_Recv (void \* buf, int count, MPI\_Datatype dtype, int fuente, int tag, MPI\_Comm comm, MPI\_Status \* estado);

buf y status son argumentos de salida; el resto son entradas.

La función devuelve un código de error

Fortran

El argumento de salida BUF es una matriz; su tipo debe coincidir con el tipo en DTYPE . Al igual que con MPI\_SEND, BUFpodría ser simplemente una variable escalar del tipo correcto.

Los argumentos de entrada COUNT , DTYPE , SOURCE , TAG , COMM son del tipo INTEGER.

El argumento de salida STATUS es una matriz INTEGER con elementos MPI\_STATUS\_SIZE.

El argumento de salida IERR es de tipo INTEGER; contiene un código de error cuando devuelve MPI\_RECV.

Algunos problemas al recibir mensajes para recordar son:

Se acepta un máximo de COUNT elementos de tipo DTYPE;

Los procesos de envío y recepción deben estar de acuerdo con el tipo de datos

Cuando esta rutina retorna, los datos del mensaje recibido se han copiado en el búfer.

### Ejemplo: Enviar y recibir

Las clasificaciones de los procesos se usan para especificar el origen y el destino. Específicamente, el proceso 0 envía un mensaje al proceso 1, y el proceso 1 lo recibe.

### Recepción de comodines

El procesador 1 solo recibirá un mensaje del procesador 0 con la etiqueta 17. Como se mencionó anteriormente, es posible configurar la recepción para que esté abierta a los mensajes de cualquier procesador o etiqueta de mensaje. Esto se llama recepción comodín.

Cuando la función MPI\_RECV se completa con algún mensaje, el proceso de destino tiene poca información al respecto.

La variable de estado es una estructura que contiene la información del mensaje como miembros. La siguiente expresión se usa para obtener la fuente de la recepción wildcarded:

status.MPI\_SOURCE, y despue Para obtener la información de la etiqueta, usa la expresión:

status.MPI\_TAG. La variable de estado es una matriz entera con la información del mensaje almacenada en sus elementos. Para obtener la fuente de la recepción comodín, use la siguiente expresión:

Status (MPI\_SOURCE) y para obtener la información de la etiqueta use:

Status (MPI\_TAG) donde nuevamente asumimos que "estado" es el nombre de la variable de estado.

### Tamaño del mensaje

El proceso de destino no conoce la siguiente información fundamental:

* El tamaño del mensaje
* La cantidad de elementos del MPI Data\_type especificado que se enviaron

La instrucción message-receive de nuestro programa de ejemplo fue:

MPI\_Recv (b,100, MPI\_DOUBLE,0,17,MPI\_COMM\_WORLD,&status);

La cantidad máxima de elementos que puede contener la matriz b , no necesariamente la cantidad de elementos realmente recibidos, el recuento de mensajes se almacena en la variable de estado y se puede extraer de él. La función MPI [auxiliar MPI\_GET\_COUNT](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1121&w=MPI_GET_COUNT#term) se usa para este propósito.

La sintaxis para MPI\_GET\_COUNT en C y Fortran se muestra a continuación.

DO:

int MPI\_Get\_count(MPI\_Status \*status, MPI\_Datatype dtype, int \*count);

Fortran:

MPI\_GET\_COUNT (STATUS, DTYPE, COUNT, IERR)

La operación de MPI\_GET\_COUNT es la misma para C o Fortran. El código debe proporcionar una variable válida MPI\_STATUS y MPI\_DATATYPE y el conteo real de los elementos en el mensaje se devuelve en la variable "conteo".

### Ejemplo de programa comodín

Aquí mostramos un programa de ejemplo que es idéntico al programa de envío / recepción de ejemplo que dimos anteriormente, con la excepción de que se usan comodines en la instrucción de recepción. Además, se utiliza MPI\_GET\_COUNT para determinar el tamaño del mensaje recibido y se han agregado las declaraciones de salida para mostrar que la recepción del comodín funcionó como se esperaba.

### Comportamiento en tiempo de ejecución

Cuando se envía un mensaje usando MPI\_SEND, puede suceder una de dos cosas: El mensaje puede copiarse en un buffer interno MPI y luego transferirse a su destino, en segundo plano, o el mensaje puede dejarse donde está, en las variables del programa, hasta que el proceso de destino esté listo para recibirlo. En ese momento, el mensaje se transfiere a su destino.

En este modelo, cuando se envía un mensaje utilizando MPI\_SEND, el mensaje se almacena en el búfer de inmediato y se entrega posteriormente de forma asincrónica, o los procesos de envío y recepción se sincronizan.

### Bloqueo y finalización

La finalización es simple pero no tan intuitivo. Una llamada a MPI\_SEND se completa cuando el mensaje especificado en la llamada se transfirió a MPI. En otras palabras, las variables pasadas a MPI\_SEND ahora se pueden sobrescribir y reutilizar. Recuerde de la sección anterior que puede haber sucedido una de dos cosas: o bien MPI copió el mensaje en un búfer interno para una entrega asincrónica posterior; o si no, MPI esperó a que el proceso de destino recibiera el mensaje. Si MPI copió el mensaje en un búfer interno, entonces la llamada a MPI\_SEND puede completarse oficialmente, aunque el mensaje aún no haya salido del proceso de envío.

4.5.10

## Envíos y recepciones sin bloqueo

### Envíos y recepciones sin bloqueo

Otra forma de invocar operaciones de envío y recepción que no bloquea el proceso de llamada. Es posible separar el inicio de una operación de envío o recepción de su finalización haciendo dos llamadas separadas a MPI. La primera llamada inicia la operación y la segunda lo completa. Si se utiliza dicha separación, se denomina comunicación sin bloqueo.

La comunicación de bloqueo y no bloqueo se puede mezclar para la misma transferencia de datos. El procesador de origen podría usar un envío de bloqueo y el proceso de destino podría usar un proceso de recepción que no sea de bloqueo, o viceversa.

### Posiciones de publicación, finalización y solicitud

Una vez que se ha publicado una operación de envío o recepción, MPI proporciona dos formas distintas de completarla. Un proceso puede probar para ver si la operación se completó sin bloqueo en la finalización. Después de publicar un envío o recepción con una llamada a una rutina que no sea de bloqueo, el proceso de [publicación](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1130&w=posting#term) necesita alguna forma de referirse a la operación publicada.

Los envíos y las recepciones pueden publicarse (iniciarse) llamando a rutinas que no sean de bloqueo.

### Publicación de envíos sin bloqueo

Un envío sin bloqueo al finalizar la operación de envío. La secuencia de llamada es similar a la secuencia de llamada para la rutina de bloqueo MPI\_SEND, pero incluye un argumento de salida adicional, un identificador de solicitud.

Algunos aspectos a tener en cuenta son:

* El cuerpo del mensaje está especificado por los primeros tres argumentos (BUF, COUNT y DTYPE en Fortran) y el sobre del mensaje por los segundos tres (DEST, TAG y COMM en Fortran).
* La fuente del mensaje.
* Cuando vuelve esta rutina.
* Se requiere otra llamada a MPI para completar la operación de envío publicada por esta rutina.
* Ninguno de los argumentos pasados ​​a MPI\_ISEND debe leerse o escribirse hasta que se complete la operación de envío.

### Publicación recibe sin bloquear

Una recepción sin bloqueo en su finalización. La secuencia de llamada es similar a la secuencia de llamada para la rutina de bloqueo MPI\_RECV, pero el argumento de estado se reemplaza por un identificador de solicitud; ambos son argumentos de salida.

Tenga en cuenta que:

* El cuerpo del mensaje está especificado por los primeros tres argumentos (BUF, COUNT y DTYPE en Fortran) y el sobre del mensaje por los segundos tres (DEST, TAG y COMM en Fortran).
* Se acepta un máximo de elementos de conteo de tipo DTYPE.
* Los procesos de envío y recepción deben estar de acuerdo con el tipo de datos; si no están de acuerdo, es un error.
* Cuando esta rutina regrese, la recepción ha sido publicada (iniciada) pero aún no ha sido completada.
* Se requiere otra llamada a MPI para completar la operación de recepción publicada por esta rutina.
* Ninguno de los argumentos pasados ​​a MPI\_IRECV debe leerse o escribirse hasta que se complete la operación de recepción.

### Finalización: espera y prueba

#### Finalización: espera y prueba

Un envío o recepción es publicado por una rutina que no se bloquea, entonces se puede verificar su estado de finalización. MPI proporciona rutinas de finalización de bloqueo y no bloqueo.

#### Finalización: Esperando

Un proceso que ha publicado un envío puede esperar posteriormente para que se complete la operación publicada

Argumentos para la rutina MPI\_WAIT son:

* Solicitud: un identificador de solicitud.
* Estado - para recibir, información sobre el mensaje recibido; para enviar, puede contener un código de error.

Además, se devuelve un código de error.

La sintaxis para MPI\_WAIT en C y Fortran es

DO:

int MPI\_Wait( MPI\_Request \*request, MPI\_Status \*status );

Se devuelve un código de error.

Fortran:

MPI\_WAIT (REQUEST, STATUS, IERR)

#### Finalización: Prueba

Un proceso que ha publicado un envío. El envío o recepción publicada se identifica al pasar un identificador de solicitud.

Los argumentos para la rutina MPI\_TEST son:

* Solicitud.
* Flag.
* Estado.

La sintaxis para MPI\_TEST en C y Fortran se da a continuación.

DO:

int MPI\_Test (MPI\_Request \*request, int \*flag, MPI\_Status \*status);

Se devuelve un código de error.

Fortran:

MPI\_TEST (REQUEST, FLAG, STATUS, IERR)

### Ventajas y desventajas

Ventajas

El uso selectivo de rutinas sin bloqueo hace que sea mucho más fácil escribir código sin punto muerto.

El proceso de destino realiza el trabajo en el bucle mientras los datos del mensaje aún están en tránsito. Una vez que llega, puede funcionar con los datos del mensaje.

MPI\_IRECV(...,request)

...

arrived=FALSE

while (arrived == FALSE) {

"work planned for processor to do while waiting for message data"

MPI\_TEST(request,arrived,status)

}

"work planned for processor to do with the message data"

### Ejemplo de envío / recepción

Este programa de ejemplo es una revisión del ejemplo anterior de envíos y recepciones El proceso 0 intenta intercambiar mensajes con el proceso 1. Cada proceso comienza publicando una recepción para un mensaje del otro. Entonces, cada proceso bloquea en un envío. Finalmente, cada proceso espera a que se complete su recepción publicada anteriormente.

Cada proceso completa su envío porque el otro proceso ha publicado una recepción coincidente. Cada proceso completa su recepción porque el otro proceso envía un mensaje que coincide. A menos que ocurra una falla del sistema, el programa se ejecuta hasta su finalización.

## Modos de envío y recepción

Hay cuatro modos de envío, pero solo un modo de recepción:

* Modo estándar Enviar.
* Envío en modo síncrono.
* Modo listo Enviar.
* Modo Buffered Enviar.

El envío de modo estándar que utilizamos en todo nuestro código de ejemplo hasta ahora en esta lección, es el más utilizado.

Cada uno de los cuatro modos de envío está disponible tanto en formas de bloqueo como de no bloqueo, lo que hace que haya ocho funciones de envío disponibles. Los envíos síncronos, en búfer y listos se indican al agregar las letras S, B y R, respectivamente, al nombre de la función.

## Ejercicio 4 - Juego de la vida

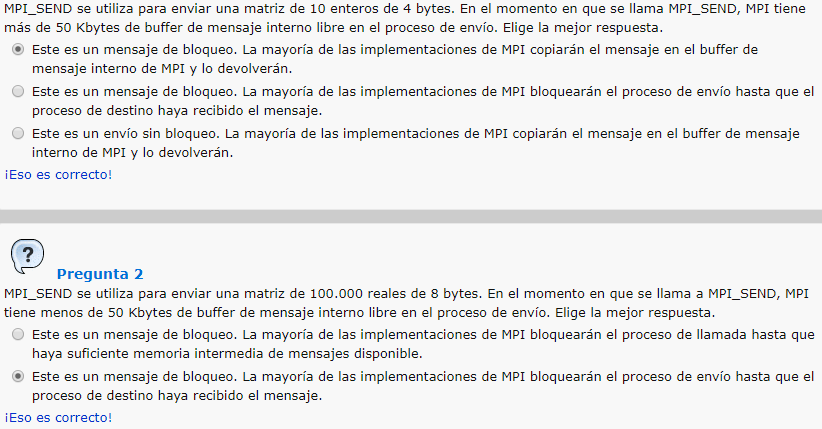
Descomposición del dominio

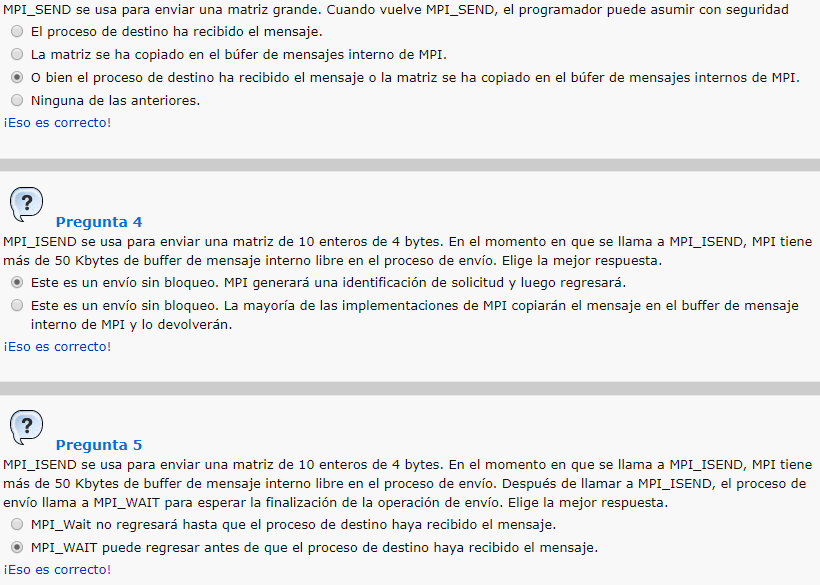
El programa "Juego de la vida" en paralelo, debemos configurar nuestra descomposición de dominio, es decir, dividir el dominio en pedazos y enviar un fragmento a cada procesador.

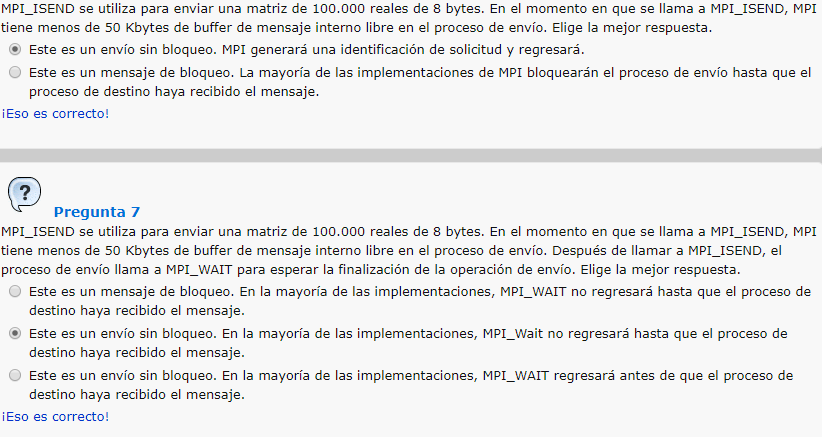
## Ejercicio 4: búsqueda paralela

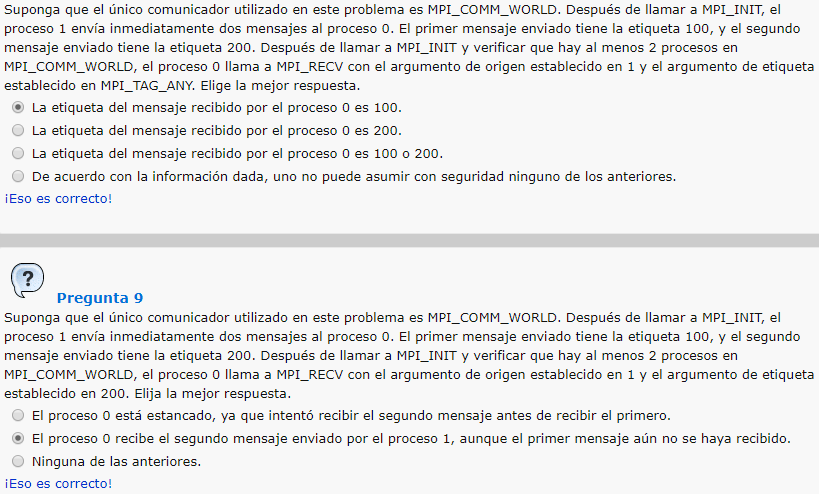
La descripción del problema es la misma que el problema inicial, que implementa una búsqueda paralela de una matriz de enteros extremadamente grande (varios miles de elementos). El programa encuentra todas las ocurrencias de un entero determinado, llamado el objetivo, y escribe todos los índices de la matriz donde se encontró el destino a un archivo de salida. Además, el programa lee tanto el valor objetivo como todos los elementos de la matriz de un archivo de entrada.

## Autocomprobación - Comunicaciones punto a punto









# Comunicación de datos no contiguos o tipos de datos mixtos

## Comunicación de datos no contiguos o tipos de datos mixtos

### Introducción

En las lecciones anteriores, aprendió a enviar y recibir mensajes donde todos los datos eran de un solo tipo intrínseco a MPI y eran contiguos en una matriz. Si bien es posible que sus datos se comporten bien, es probable que necesite transmitir colecciones de datos de tipos mixtos o datos dispersos dentro de una matriz.

En Fortran, las columnas son contiguas, es decir, x (5,7) es "al lado de" x (6,7). Aunque los datos en la memoria a menudo se organizan de la misma manera, no siempre es el caso.

### Objetivos

En esta lección, aprenderá acerca de las siguientes estrategias y características en MPI que le permiten transmitir colecciones de datos de tipos mixtos o datos dispersos dentro de una matriz:

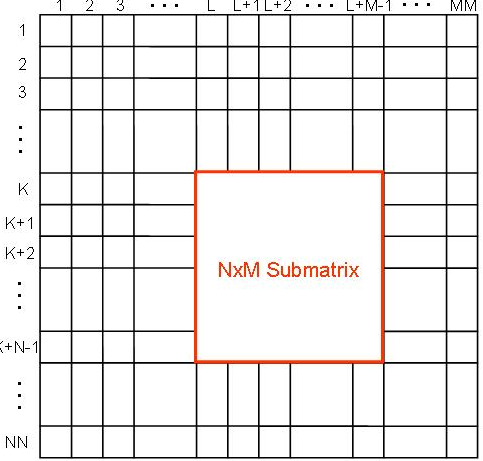
* Envío de mensajes múltiples
* Buffering
* Empacar y desempacar
* Tipos de datos derivados

Al final de esta lección, podrá escribir un programa MPI utilizando estas técnicas para transmitir colecciones de datos dentro de una matriz. Se proporcionan dos ejercicios y una auto prueba para que pueda practicar lo que ha aprendido.

## Envío de mensajes múltiples

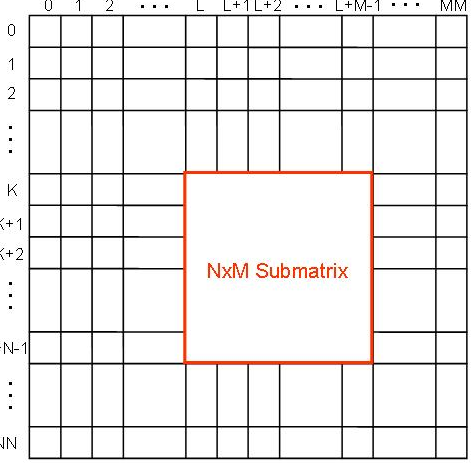
El enfoque más simple para comunicar tipos de datos mixtos y datos no contiguos es identificar las piezas más grandes de sus datos que cumplen individualmente los requisitos de ser de tipo intrínseco homogéneo y contiguo, y enviar cada una de esas piezas como un mensaje separado.

Las matrices se almacenan de modo que los elementos sucesivos de una columna sean contiguos, con el último elemento en una columna seguido del primer elemento de la columna siguiente. Como no está enviando columnas completas de la matriz original, las columnas de su submatriz no serán contiguas. Sin embargo, los elementos dentro de una columna serán contiguos, por lo que puede enviar un mensaje para cada columna de la submatriz.



En C, el enfoque sería similar, pero como los elementos de las filas son contiguos en lugar de columnas, enviaría N mensajes que contienen los M elementos en cada fila de la submatriz, mientras que en el ejemplo de Fortran enviaría M mensajes que contenían los N elementos de una columna.

La siguiente matriz es similar, pero comienza de 0 y no en 1.



El envío correspondiente podría verse así:

for (i=0; i<n; ++i) {

MPI\_Send(&a[k+i][l], m, MPI\_DOUBLE, dest, tag, MPI\_COMM\_WORLD);

}

En cualquier idioma, si el procesador receptor no conoce los valores de N , M , K o L , se pueden enviar en un mensaje separado.

## Almacenamiento en búfer

### Almacenamiento en búfer

Cuando los datos que necesita para comunicarse no son contiguos, un enfoque podría ser copiarlos en un búfer contiguo.

Fortran:

icount = 0

do j = l, l+m-1

do i = k, k+n-1

icount = icount + 1

p(icount) = a(i,j)

enddo

enddo

call MPI\_Send(p, n\*m, MPI\_DOUBLE, dest, tag, MPI\_COMM\_WORLD, ierr)

DO:

p = &buffer;

for (i=k; i<k+n; ++i) {

for(j=l; j<l+m; ++j) {

\*(p++) = a[i][j];

}

}

MPI\_Send(p, n\*m, MPI\_DOUBLE, dest, tag, MPI\_COMM\_WORLD)

Copiar en un búfer elimina la cantidad excesiva de mensajes generados por el enfoque de mensajes múltiples a costa de memoria adicional para el búfer y el tiempo de CPU adicional para realizar la copia en el búfer.

### Una manera incorrecta y tentadora de ampliar el almacenamiento en búfer

A menudo es posible codificar los valores de un tipo como valores de otro tipo. En nuestro ejemplo de submatriz, podríamos convertir los valores de N , M , K y L en coma flotante para incluirlos en nuestro búfer n este punto, puede tener la tentación de usar un truco de programación (por ejemplo, EQUIVALENCIA, la función TRANSFERENCIA o el tipo de puntero) para poner los patrones de bits para valores de un tipo en un búfer declarado de otro tipo . Este enfoque puede ser muy peligroso. El problema fundamental con este enfoque es que MPI transmite valores, no solo patrones de bits.

### Almacenamiento en búfer de la manera correcta: empacar y desempaquetar

La forma correcta de llenar un buffer es usar la rutina **MPI\_PACK** . Llama a MPI\_PACK con argumentos que describen el búfer que está rellenando y la mayoría de los argumentos que habría proporcionado al usar MPI\_SEND. MPI\_PACK copia sus datos en el búfer y, si es necesario, los traduce en una representación intermedia estándar. Una vez que MPI\_PACK ha colocado todos los datos que desea enviar en el búfer, puede enviar el búfer (dando su tipo como MPI\_PACKED) y no se realizarán más traducciones.

Con MPI\_PACK, el ejemplo de la submatriz Fortran se convierte en:

icount = 0

do i = 1, m

call MPI\_PACK(a(k,l+i-1), n, MPI\_DOUBLE, buffer, bufsize, &

icount, MPI\_COMM\_WORLD, ierr)

enddo

call MPI\_SEND(buffer, icount, MPI\_PACKED, dest, tag, MPI\_COMM\_WORLD, ierr)

y en C, se convierte en:

count = 0;

for(i=0; i<n; i++){

MPI\_Pack(&a[k+i][l], m, MPI\_DOUBLE, buffer, bufsize,

&count, MPI\_COMM\_WORLD);

}

MPI\_Send(buffer, count, MPI\_PACKED, dest, tag, MPI\_COMM\_WORLD);

## Tipos de datos derivados

### Tipos de datos derivados

MPI tiene un enfoque más eficiente para manejar los tipos de datos mixtos y los datos no contiguos que los dos enfoques que acabamos de discutir, el envío de múltiples mensajes y el almacenamiento en memoria intermedia (empaque y desempaquetado), llamados **tipos de datos derivados**.

El embalaje y desembalaje se puede realizar directamente desde y hacia sus búferes de comunicaciones internas, lo que generalmente hace que su programa sea más eficiente al eliminar la necesidad de:

* El buffer intermedio explícito que se usa cuando haces el embalaje y desempaque
* La copia entre el buffer intermedio y el buffer de comunicaciones

Hay varias rutinas MPI que se pueden usar para crear tipos de datos derivados. Para nuestro ejemplo, usaremos **MPI\_TYPE\_VECTOR**, que crea un tipo de datos que consiste en bloques de datos espaciados uniformemente, cada bloque tiene el mismo tamaño.

call MPI\_TYPE\_VECTOR(m, n, nn, MPI\_DOUBLE, my\_mpi\_type, ierr)

call MPI\_TYPE\_COMMIT(my\_mpi\_type, ierr)

call MPI\_SEND(a(k,l), 1, my\_mpi\_type, dest, tag, MPI\_COMM\_WORLD, ierr)

En C, la idea es similar, excepto que los segmentos de filas servirán como bloques en lugar de segmentos de columnas, ya que C almacena la matriz en orden de fila mayor:

MPI\_Type\_vector(n,m, mm, MPI\_DOUBLE, &my\_mpi\_type);

MPI\_Type\_commit(&my\_mpi\_type);

MPI\_Send(&a[k][l], 1, my\_mpi\_type, dest, tag, MPI\_COMM\_WORLD);

### Tipos de datos definidos por el usuario

Crear un tipo de datos derivado para usarlo solo una vez antes de liberarlo puede ser un poco desperdiciado. Es mucho más efectivo crear tipos de datos derivados que describan patrones recurrentes de acceso y luego reutilizarlos para cada aparición de ese patrón de acceso. El ejemplo clásico de esto es el uso de un tipo de datos derivado para describir un tipo de datos definido por el usuario en el idioma que está utilizando. Esta técnica se llama **mapeo**.

Aquí hay un ejemplo simple de mapeo de un tipo de estructura C:

struct SparseElt { /\* representation of a sparse matrix element \*/

int location[2]; /\* where the element belongs in the overall matrix \*/

double value; /\* the value of the element \*/

};

/\* a representative variable of this type \*/

struct SparseElt anElement;

/\* length, displacement, and type arrays used to describe an MPI derived type \*/

/\* their size reflects the number of components in SparseElt \*/

int lena[2];

MPI\_Aint loca[2];

MPI\_Datatype typa[2];

MPI\_Aint baseaddress;

/\* a variable to hold the MPI type indicator for SparseElt \*/

MPI\_Datatype MPI\_SparseElt;

/\* set up the MPI description of SparseElt \*/

MPI\_Address(&anElement, &baseaddress);

lena[0] = 2; /\* anElement.location has length of 2 ints\*/

MPI\_Address(&anElement.location,&loca[0]);

loca[0] -= baseaddress; /\* byte address relative to start of structure \*/

typa[0] = MPI\_INT;

lena[1] = 1; /\* anElement.value has length of 1 double\*/

MPI\_Address(&anElement.value ,&loca[1]);

loca[1] -= baseaddress;

typa[1] = MPI\_DOUBLE;

MPI\_Type\_struct(2, lena, loca, typa, &MPI\_SparseElt);

MPI\_Type\_commit(&MPI\_SparseElt);

En este ejemplo, construimos tres arreglos que contienen la longitud, la ubicación y los tipos de los componentes que se transferirán cuando se use este tipo. Luego restamos la dirección de la variable completa de las direcciones de sus componentes;

integer, parameter :: rk = kind(1.0d0)

integer :: baseaddress, mpi\_sparselt, ier

!... length, displacement, and type arrays used to describe an MPI derived type

!... their size reflects the number of components in SparseElt

integer, dimension(2) :: lena, loca, typa

!... representation of a sparse matrix element

type sparselt

integer :: location(2) !...where the element belongs in the overall matrix

real(rk) :: value !... the value of the element

end type sparselt

!... a representative variable of this type

type(sparselt) anelement

!... set up the MPI description of SparseElt

call mpi\_address(anelement, baseaddress, ier)

lena(1)= 2 !... anelement%location has length of 2 integers

call mpi\_address(anelement%location, loca(1), ier)

loca(1) = loca(1) - baseaddress; !... byte address relative to start of type

typa(1) = MPI\_INTEGER

lena(2) = 1 !... anelement%value has length of 1 rk

call mpi\_address(anelement%value, loca(2), ier)

loca(2) = loca(2) - baseaddress

typa(2) = MPI\_DOUBLE\_PRECISION

call mpi\_type\_struct(2, lena, loca, typa, mpi\_sparselt, ier)

call mpi\_type\_commit(mpi\_sparselt, ier)

### Construir tipos de datos derivados

Ya hemos mostrado el uso de MPI\_TYPE\_VECTOR para construir tipos de datos derivados. La forma más general de construir un tipo de datos derivado es mediante el uso de MPI\_TYPE\_STRUCT, que permite especificar la longitud, la ubicación y el tipo de cada componente de manera independiente. Hay varios otros procedimientos disponibles para describir patrones comunes de acceso, principalmente dentro de matrices. Estos son:

* MPI\_TYPE\_CONTIGUOUS
* MPI\_TYPE\_HVECTOR
* MPI\_TYPE\_INDEXED
* MPI\_TYPE\_HINDEXED

[**MPI\_TYPE\_CONTIGUOUS**](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1149&w=MPI_TYPE_CONTIGUOUS#term) es el más simple de estos, que describe una secuencia contigua de valores en la memoria. Por ejemplo, usándolo de la siguiente manera en C:

MPI\_Type\_contiguous(2,MPI\_DOUBLE,&MPI\_2D\_POINT);

MPI\_Type\_contiguous(3,MPI\_DOUBLE,&MPI\_3D\_POINT);

Crea nuevos indicadores de tipo MPI\_2D\_POINT y MPI\_3D\_POINT. Estos indicadores de tipo te permiten tratar pares consecutivos de dobles como coordenadas de puntos en un espacio de 2 dimensiones y secuencias de tres dobles como coordenadas de puntos en un espacio tridimensional.

En Fortran, las llamadas solo difieren por el argumento de error:

call mpi\_type\_contiguous(2,mpi\_double,mpi\_2d\_point,ierr)

call mpi\_type\_contiguous(3,mpi\_double,mpi\_3d\_point,ierr)

[**MPI\_TYPE\_HVECTOR**](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1149&w=MPI_TYPE_HVECTOR#term) es similar a MPI\_TYPE\_VECTOR, excepto que la distancia entre bloques sucesivos se especifica en bytes en lugar de elementos. La razón más común para especificar esta distancia en bytes sería que los elementos de algún otro tipo se intercalan en la memoria con los elementos de interés. Por ejemplo, si tiene una matriz de tipo SparseElt, puede usar

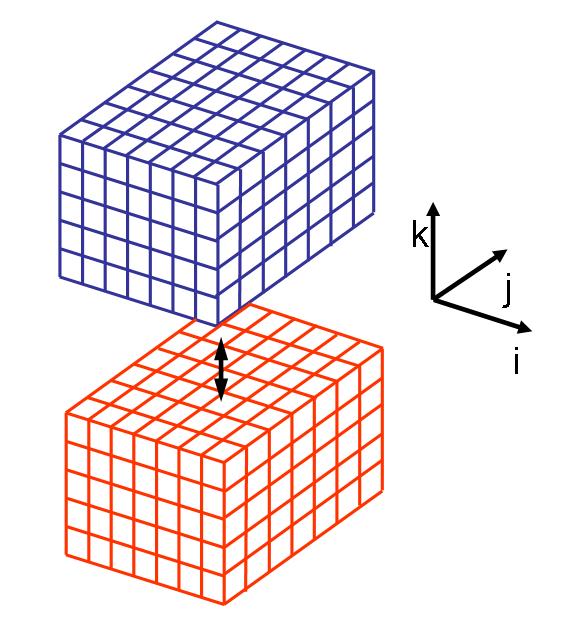
* [MPI\_TYPE\_HVECTOR](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1149&w=MPI_TYPE_HVECTOR#term)para describir la matriz de componentes de valor.
* [MPI\_TYPE\_INDEXED](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1149&w=MPI_TYPE_INDEXED#term) describe secuencias que pueden variar tanto en longitud como en espaciado. Debido a que la ubicación de estas secuencias se mide en elementos en lugar de bytes, es más apropiado para identificar partes arbitrarias de una sola matriz.
* [MPI\_TYPE\_HINDEXED](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1149&w=MPI_TYPE_HINDEXED#term) es similar a MPI\_TYPE\_INDEXED excepto que las ubicaciones se especifican en bytes en lugar de elementos. Permite la identificación de partes arbitrarias de matrices arbitrarias, sujetas únicamente al requisito de que todas tengan el mismo tipo.

### Coincidencia de mensajes y desajuste

De la misma forma que no hay nada en el contenido de un mensaje que indique si se construyó con MPI\_PACK, no hay nada que indique si se han utilizado o no tipos de datos derivados en su construcción. Todo lo que importa es que el emisor y el receptor estén de acuerdo con la naturaleza de la secuencia de valores primitivos que representa el mensaje. Por lo tanto, un mensaje construido y enviado a través de MPI\_PACK podría ser recibida usando tipos de datos derivados, o un mensaje enviado usando tipos de datos derivados podría ser recibida como MPI\_PACKED y distribuye mediante MPI\_UNPACK. De forma similar, un mensaje podría enviarse usando un tipo de datos derivado y ser recibido usando un tipo de datos derivado diferente.

### Combinación de tipos de datos derivados

A menudo es conveniente construir un tipo de datos derivado combinando otros tipos de datos derivados. El ejemplo que se presenta aquí proviene de un código utilizado para simular flujos en turbomáquinas (motores a reacción, turbopunzas de cohetes, etc.). En este código y muchos solucionadores de flujo similares, se genera una cuadrícula tridimensional para discretizar el dominio, con cada triple ordenada de índices (i, j, k) que representa una ubicación en el espacio. Se resuelven cinco ecuaciones acopladas (ecuaciones de Navier-Stokes), por lo que en cada punto de la cuadrícula hay que calcular cinco valores. La solución se guarda comúnmente en una matriz de rango 4, con los primeros tres índices representando la ubicación espacial, y el último índice representando los cinco valores, q (i, j, k, n).

  
Descomposición del dominio de la dirección k.

Después de que cada procesador complete su parte del cálculo, los datos en la superficie de la interfaz entre los procesadores deben intercambiarse, y aquí es donde los tipos de datos derivados son útiles. Primero, crearemos un tipo de datos derivado que contenga todos los valores *i* y *j* para una *k* especificada y una *n* especificada. Luego combinaremos cinco de estos tipos de datos derivados, uno para cada *n*, en un nuevo tipo de datos derivado.

Primero creamos un tipo de datos derivado que contiene los valores en *q* para todo *i*, todos *j*, un valor único de *k* y un valor único de *n*. Se puede pensar que esto crea un tipo con "rayas" *jmax*, cada una con valores *imax*, que cubre una superficie *ij* a una constante *k*. Este tipo intermedio ha sido llamado *q\_elem\_type* ya que representa todos los valores *i* y *j* para un solo elemento, o valor *n* individual, de *q*.

CALL MPI\_TYPE\_VECTOR(jmax, imax, imax, mpi\_real, q\_elem\_type, IERR)

Como revisión, los argumentos son:

* *jmax* - número de bloques de datos, en este caso, número de rayas a lo largo de la dirección i
* *imax* - número de valores en cada banda
* *imax* - paso entre rayas, en unidades del siguiente argumento, es decir, número de valores MPI\_REAL
* *mpi\_real* - tipo de datos que va con el argumento de zancada
* *q\_elem\_type* - nombre del nuevo tipo de datos derivado
* *IERR* - código de error (solo Fortran)

Ahora necesitamos combinar cinco instancias de *q\_elem\_type*, una para cada valor de *n* . En el tipo de datos derivado anterior, la zancada era valores *imax*, cada valor era un MPI\_REAL. Para el tipo actual, definiremos la zancada en **bytes**, que es la distancia entre diferentes valores del índice *n* en la matriz *q* :

istriden = imax\*jmax\*kmax\*nbytes\_per\_word

Donde *imax*, *jmax* y *kmax* son las tres primeras dimensiones de la matriz *q* y *nbytes\_per\_word* es (sorprendentemente) el número de bytes por palabra.

La función MPI\_TYPE\_HVECTOR es exactamente igual a MPI\_TYPE\_VECTOR, excepto que la zancada se da en bytes en lugar de como un número del tipo especificado:

CALL MPI\_TYPE\_HVECTOR (5, 1, istriden, q\_elem\_type, kplane\_type, IERR)

Aquí, estamos creando un tipo de datos derivado que contiene cinco bloques de *q\_elem\_type*, cada bloque contiene una sola instancia, con una zancada de bytes *istriden* entre cada uno. El nuevo tipo de datos derivado se llama *kplane\_type*.

Finalmente, asignaremos el nuevo tipo para que podamos usarlo en el envío de mensajes:

CALL MPI\_TYPE\_COMMIT (kplane\_type, IERR)

El tipo definido por el usuario *q\_elem\_type* nunca se *confirmó*. Solo es necesario confirmar un tipo si se usa en llamadas a rutinas de paso de mensajes. Si simplemente se usa como un componente de otro tipo derivado, no tiene que ser confirmado.

En C, este ejemplo no requiere una combinación de tipos de datos derivados, ya que las matrices se almacenan en el orden opuesto al de Fortran. No hay nada inherentemente más simple en el uso de C con MPI en lugar de Fortran; este ejemplo se construyó para ilustrar los tipos de datos derivados combinados en Fortran. El caso análogo (combinación de tipos de datos derivados) en C tendría los índices de matriz invertidos, *q [n] [k] [j] [i]*. Los tipos de datos derivados se construirían de la siguiente manera:

MPI\_Type\_vector(jmax, imax, imax, MPI\_FLOAT, &q\_elem\_type);

istriden = imax\*jmax\*kmax\*nbytes\_per\_word;

MPI\_Type\_hvector(5, 1, istriden, q\_elem\_type, &kplane\_type);

MPI\_Type\_commit(&kplane\_type);

### Obtención de información sobre sus tipos derivados

Una vez que haya definido un tipo de datos derivado, existen varios procedimientos de utilidad que puede usar para obtener información al respecto. Son:

* [MPI\_TYPE\_LB](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1152&w=MPI_TYPE_LB#term) y[MPI\_TYPE\_UB](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1152&w=MPI_TYPE_UB#term) , que proporcionan los límites inferior y superior del tipo de datos.
* [MPI\_TYPE\_EXTENT](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1152&w=MPI_TYPE_EXTENT#term) , que proporciona la extensión del tipo de datos. En la mayoría de los casos, esta es la cantidad de memoria que ocupará un valor del tipo de datos.
* [MPI\_TYPE\_SIZE](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1152&w=MPI_TYPE_SIZE#term) , que proporciona el tamaño del tipo de datos en un mensaje. Si el tipo de datos está disperso en la memoria, esto puede ser significativamente más pequeño que la extensión del tipo de datos.

## Ejercicio 5 - Juego de la vida

La capacidad de crear su propio tipo de datos para transferir datos fácilmente con un patrón específico de acceso a la memoria, incluidos datos de diversos tipos. En este ejercicio, usará un tipo de datos derivados para generalizar la descomposición del dominio utilizada en el problema del curso anterior "Juego de la vida" donde se eligió la descomposición del dominio de modo que se accediera contiguamente a los elementos de la matriz. Ahora modificará el código para manejar elementos de matriz no contiguos.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Conway Game of Life

2 processors

divide domain left-right

(break with vertical line)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include "mpi.h"

#include <stdio.h>

#include <stdlib.h>

#define NI 200

#define NJ 200

#define NSTEPS 500

void main(int argc, char \*argv[]){

int i, j, n, im, ip, jm, jp, nsum, isum, isum1, nprocs, myid;

int ig, jg, i1g, i2g, j1g, j2g, ninom, njnom, ninj, i1, i2, i2m,

j1, j2, j2m, ni, nj;

int niproc, njproc;

int \*\*old, \*\*new, \*old1d, \*new1d;

MPI\_Status status;

**MPI\_Datatype column\_type;**

float x;

/\* initialize MPI \*/

MPI\_Init(&argc,&argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&nprocs);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&myid);

**/\* nominal number of points per proc. in each direction,**

**without ghost cells, assume numbers divide evenly \*/**

**niproc = 1;**

**njproc = nprocs; /\* divide domain in j direction only \*/**

ninom = NI/niproc;

njnom = NJ/njproc;

/\* global starting and ending indices (without ghost cells) \*/

**i1g = 1;**

**i2g = ninom;**

**j1g = (myid\*njnom) + 1;**

**j2g = j1g + njnom - 1;**

/\* local starting and ending indices, including ghost cells \*/

i1 = 0;

i2 = ninom + 1;

i2m = i2 - 1;

j1 = 0;

**j2 = njnom + 1;**

j2m = j2 - 1;

/\* allocate arrays; want elements to be contiguous, so

allocate 1-D arrays, then set pointer to each row (old

and new) to allow use of array notation for convenience \*/

ni = i2-i1+1;

nj = j2-j1+1;

ninj = ni\*nj;

old1d = malloc(ninj\*sizeof(int));

new1d = malloc(ninj\*sizeof(int));

old = malloc(ni\*sizeof(int\*));

new = malloc(ni\*sizeof(int\*));

for(i=0; i<ni; i++){

old[i] = &old1d[i\*nj];

new[i] = &new1d[i\*nj];

}

/\* Initialize elements of old to 0 or 1.

We're doing some sleight of hand here to make sure we

initialize to the same values as in the serial case.

The rand() function is called for every i and j, even

if they are not on the current processor, to get the same

random distribution as the serial case, but they are

only used if i and j reside on the current procesor. \*/

for(ig=1; ig<=NI; ig++){

for(jg=1; jg<=NJ; jg++){

x = rand()/((float)RAND\_MAX + 1);

**/\* if this j is on the current processor \*/**

**if( jg >= j1g && jg <= j2g ){**

/\* local i and j indices, accounting for lower ghost cell \*/

**i = ig;**

**j = jg - j1g + 1;**

if(x<0.5){

old[i][j] = 0;

}else{

old[i][j] = 1;

}

}

}

}

**/\* Create derived type for single column of array.**

**There are NI "blocks," each containing 1 element,**

**with a stride of nj between the blocks \*/**

**MPI\_Type\_vector(NI, 1, nj, MPI\_INT, &column\_type);**

**MPI\_Type\_commit(&column\_type);**

/\* iterate \*/

for(n=0; n<NSTEPS; n++){

/\* transfer data to ghost cells \*/

if(nprocs == 1){

/\* left and right columns \*/

for(i=1; i<i2; i++){

old[i][0] = old[i][j2m];

old[i][j2] = old[i][1];

}

/\* top and bottom \*/

for(j=1; j<j2; j++){

old[0][j] = old[i2m][j];

old[i2][j] = old[1][j];

}

/\* corners \*/

old[0][0] = old[i2m][j2m];

old[0][j2] = old[i2m][1];

old[i2][j2] = old[1][1];

old[i2][0] = old[1][j2m];

}else{

if(myid == 0){

**/\* use derived type "column\_type" to transfer columns \*/**

**MPI\_Send(&old[1][j2-1], 1, column\_type, 1, 0, MPI\_COMM\_WORLD);**

**MPI\_Recv(&old[1][j2], 1, column\_type, 1, 1, MPI\_COMM\_WORLD, &status);**

**MPI\_Send(&old[1][1], 1, column\_type, 1, 2, MPI\_COMM\_WORLD);**

**MPI\_Recv(&old[1][0], 1, column\_type, 1, 3, MPI\_COMM\_WORLD, &status);**

**/\* top and bottom \*/**

**for(j=0; j<nj; j++){**

**old[0][j] = old[i2m][j];**

**old[i2][j] = old[1][j];**

**}**

**/\* corners \*/**

**MPI\_Send(&old[1][1], 1, MPI\_INT, 1, 10, MPI\_COMM\_WORLD);**

**MPI\_Recv(&old[0][0], 1, MPI\_INT, 1, 11, MPI\_COMM\_WORLD, &status);**

**MPI\_Send(&old[i2m][1], 1, MPI\_INT, 1, 12, MPI\_COMM\_WORLD);**

**MPI\_Recv(&old[i2][0], 1, MPI\_INT, 1, 13, MPI\_COMM\_WORLD, &status);**

**}else{**

**/\* use derived type "column\_type" to transfer columns \*/**

**MPI\_Recv(&old[1][0], 1, column\_type, 0, 0, MPI\_COMM\_WORLD, &status);**

**MPI\_Send(&old[1][1], 1, column\_type, 0, 1, MPI\_COMM\_WORLD);**

**MPI\_Recv(&old[1][j2], 1, column\_type, 0, 2, MPI\_COMM\_WORLD, &status);**

**MPI\_Send(&old[1][j2-1], 1, column\_type, 0, 3, MPI\_COMM\_WORLD);**

**/\* top and bottom \*/**

**for(j=0; j<nj; j++){**

**old[0][j] = old[i2m][j];**

**old[i2][j] = old[1][j];**

**}**

**/\* corners \*/**

**MPI\_Recv(&old[i2][j2], 1, MPI\_INT, 0, 10, MPI\_COMM\_WORLD, &status);**

**MPI\_Send(&old[i2m][j2m], 1, MPI\_INT, 0, 11, MPI\_COMM\_WORLD);**

**MPI\_Recv(&old[0][j2], 1, MPI\_INT, 0, 12, MPI\_COMM\_WORLD, &status);**

**MPI\_Send(&old[1][j2m], 1, MPI\_INT, 0, 13, MPI\_COMM\_WORLD);**

}

}

/\* update states of cells \*/

for(i=1; i<i2; i++){

for(j=1; j<j2; j++){

im = i-1;

ip = i+1;

jm = j-1;

jp = j+1;

nsum = old[im][jp] + old[i][jp] + old[ip][jp]

+ old[im][j ] + old[ip][j ]

+ old[im][jm] + old[i][jm] + old[ip][jm];

switch(nsum){

case 3:

new[i][j] = 1;

break;

case 2:

new[i][j] = old[i][j];

break;

default:

new[i][j] = 0;

}

}

}

/\* copy new state into old state \*/

for(i=1; i<**i2**; i++){

for(j=**1**; j<**j2**; j++){

old[i][j] = new[i][j];

}

}

}

/\* Iterations are done; sum the number of live cells \*/

isum = 0;

for(i=1; i<i2; i++){

for(j=1; j<j2; j++){

isum = isum + new[i][j];

}

}

/\* Print final number of live cells. For multiple processors,

must reduce partial sums \*/

if(nprocs > 1){

if(myid == 0){

MPI\_Recv(&isum1, 1, MPI\_INT, 1, 10, MPI\_COMM\_WORLD, &status);

isum = isum + isum1;

}else{

MPI\_Send(&isum, 1, MPI\_INT, 0, 10, MPI\_COMM\_WORLD);

}

}

if(myid == 0) printf("Number of live cells = %d\n", isum);

MPI\_Finalize();

}

## Ejercicio 5: búsqueda paralela

La capacidad de crear su propio tipo de datos para transferir datos fácilmente con un patrón específico de acceso a la memoria, incluidos datos de diversos tipos. En este ejercicio, modificará el problema de búsqueda paralela para que cada esclavo envíe un entero y un valor real al maestro. Creará y usará un tipo de datos derivado para enviar / recibir ambos datos como una sola entidad.

### Ejercicio

El problema del curso aún implementa una búsqueda paralela de una matriz de enteros. El programa debe encontrar todas las ocurrencias de un cierto entero que se llamará el objetivo. Luego debe calcular el promedio del valor objetivo y su índice. Tanto la ubicación del objetivo como el promedio se deben escribir en un archivo de salida. Además, el programa debe leer tanto el valor objetivo como todos los elementos de la matriz de un archivo de entrada.

Estos son los resultados obtenidos en el archivo dfound.data:

P 1, 62, 36.

P 2, 183, 96.5

P 3, 271, 140.5

P 3, 291, 150.5

P 3, 296, 153.

# Comunicaciones colectivas

## Comunicaciones colectivas

Implica el envío y recepción de datos entre procesos. En general, todo el movimiento de datos entre los procesos se puede lograr usando rutinas de envío y recepción de MPI.

Las rutinas de comunicación colectiva transmiten datos entre todos los procesos en un grupo. Debe asegurarse de que todos los procesadores ejecutan una llamada de comunicación colectiva determinada.

### Objetivos

En esta lección, aprenderá sobre las rutinas de comunicación colectiva que MPI proporciona para las siguientes operaciones:

* Sincronización de barreras en todos los procesos.
* Transmisión de un proceso a todos los demás procesos
* Operaciones globales de reducción tales como suma, mínimo, máximo o reducciones definidas por el usuario
* Recopila datos de todos los procesos en un solo proceso
* Distribuir datos de un proceso a todos los procesos

Operaciones avanzadas donde todos los procesos reciben el mismo resultado de una recopilación, dispersión o reducción. También hay una variante de vector de la mayoría de las operaciones colectivas donde cada mensaje puede tener un tamaño diferente en diferentes procesos.

## Sincronización de barreras

[MPI\_BARRIER](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1157&w=MPI_BARRIER#term) bloquea el proceso de llamada hasta que todos los procesos del grupo hayan llamado a la función. Cuando MPI\_BARRIER regresa, todos los procesos se sincronizan en la barrera. Este también se realiza en software y puede generar una sobrecarga considerable en algunas máquinas. En general, solo debe insertar barreras cuando realmente las necesite.

Las llamadas a MPI\_BARRIER en C y Fortran son las siguientes:

DO:

int MPI\_Barrier ( comm )

MPI\_Comm comm

Fortran:

MPI\_BARRIER ( COMM, ERROR )

INTEGER COMM, ERROR

## Transmisión

En [MPI\_BCAST le](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1158&w=MPI_BCAST#term) permite copiar datos de la memoria del procesador raíz a las mismas ubicaciones de memoria para otros procesadores en el comunicador.

Ejemplos en C y FORTRAN

C Ejemplo:

#include <mpi.h>

void main(int argc, char \*argv[]) {

int rank;

double param;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

if(rank==5) param=23.0;

MPI\_Bcast(¶m,1,MPI\_DOUBLE,5,MPI\_COMM\_WORLD);

printf("P:%d after broadcast parameter is %f \n",rank,param);

MPI\_Finalize();

}

Ejemplo de Fortran:

PROGRAM broadcast

INCLUDE 'mpif.h'

INTEGER err, rank, size

real param

CALL MPI\_INIT(err)

CALL MPI\_COMM\_RANK(MPI\_COMM\_WORLD, rank, err)

CALL MPI\_COMM\_SIZE(MPI\_COMM\_WORLD, size, err)

if(rank.eq.5) param=23.0

call MPI\_BCAST(param,1,MPI\_REAL,5,MPI\_COMM\_WORLD,err)

print \*,'P:',rank,' after broadcast param is ',param

CALL MPI\_FINALIZE(err)

END PROGRAM broadcast

## Reducción

[MPI\_REDUCE le](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1159&w=MPI_REDUCE#term) permite:

* Recoge datos de cada procesador
* Reduce estos datos a un solo valor (como una suma o un máximo)
* Almacenar el resultado reducido en el procesador raíz

La secuencia de llamada es:

MPI\_Reduce (send\_buffer, recv\_buffer, count, data\_type, reduction\_operation, rank\_of\_receiving\_process, Communicator)

C Ejemplo:

#include <stdio.h>

#include <mpi.h>

void main(int argc, char \*argv[]){

int rank;

int source,result,root;

/\* run on 10 processors \*/

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

root=7;

source=rank+1;

MPI\_Barrier(MPI\_COMM\_WORLD);

MPI\_Reduce(&source,&result,1,MPI\_INT,MPI\_PROD,root,MPI\_COMM\_WORLD);

if(rank==root) printf("P:%d MPI\_PROD result is %d \n",rank,result);

MPI\_Finalize();

}

## Reunir

[MPI\_GATHER](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1160&w=MPI_GATHER#term) es una comunicación de uno a uno . MPI\_GATHER tiene los mismos argumentos que las rutinas de dispersión coincidentes. Los argumentos de recepción solo son significativos para el proceso raíz. La recopilación también podría realizarse mediante cada proceso que llame a MPI\_SEND y el proceso raíz que llama a MPI\_RECV Nveces para recibir todos los mensajes. Un ejemplo en C es el siguiente

C Ejemplo:

#include <stdio.h>

#include <mpi.h>

void main(int argc, char \*argv[])

{

int rank,size;

double param[16],mine;

int sndcnt,rcvcnt;

int i;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&size);

sndcnt=1;

mine=23.0+rank;

if(rank==7) rcvcnt=1;

MPI\_Gather(&mine,sndcnt,MPI\_DOUBLE,param,rcvcnt,MPI\_DOUBLE,7,MPI\_COMM\_WORLD);

if(rank==7)

for(i=0;i<size;++i) printf("PE:%d param[%d] is %f \n",rank,i,param[i]]);

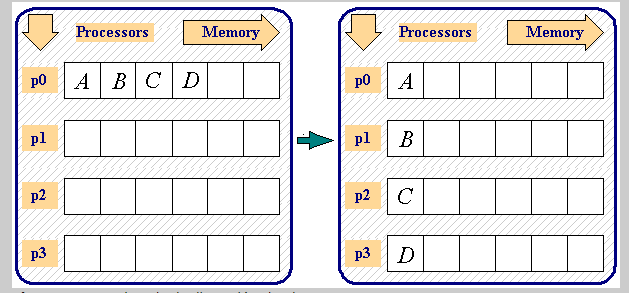
MPI\_Finalize();

}

## Dispersión

MPI Scater, es el proceso raíz que divide un conjunto de ubicaciones de la memoria contigua en partes iguales, además de enviar fragmentos a cada procesador. Su resultado es el mismo, si la raíz se ejecutara *N MPI SEND*  operaciones y cada proceso ejecutara MPI\_RECV. Los argumentos solo son significativos para el proceso de raíz.

En el siguiente ejemplo se muestran 4 valores que contienen datos contiguos, elementos de procesador o que comienza en *A* son copiados a cada procesador en la posición *A*.



## Operaciones avanzadas

Estas son algunas rutinas avanzadas de comunicación colectiva.

* [MPI\_ALLREDUCE](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1162&w=MPI_ALLREDUCE#term) : se utiliza para combinar los elementos del búfer de entrada de cada proceso y almacena el valor combinado en el búfer de recepción de todos los miembros del grupo. Operaciones de reducción definidas por el usuario: permite definir la reducción como una operación arbitraria.

Operaciones de Vector de [Agrupamiento](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1162&w=MPI_GATHERV#term) / Dispersión

* [MPI\_GATHERV](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1162&w=MPI_GATHERV#term) y [MPI\_SCATTERV](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1162&w=MPI_SCATTERV#term) permiten un conteo variable de datos desde / hacia cada proceso. Otras variaciones de [agrupamiento](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1162&w=MPI_ALLGATHER#term) / dispersión: [MPI\_ALLGATHER](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1162&w=MPI_ALLGATHER#term) y [MPI\_ALLTOALL](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1162&w=MPI_ALLTOALL#term).

No se especificó ningún proceso de raíz: todos los procesos se recopilan o dispersan.

Los argumentos de envío y recepción son significativos para todos los procesos.

* [MPI\_SCAN](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1162&w=MPI_SCAN#term) : se usa para llevar a cabo una reducción de prefijo en los datos en todo el grupo y devuelve la reducción de los valores de todos los procesos.
* [MPI\_REDUCE\_SCATTER](https://www.citutor.org/mods/_core/glossary/index.php?g_cid=1162&w=MPI_REDUCE_SCATTER#term) combina un MPI\_REDUCE y un MPI\_SCATTERV.

## Ejercicio 6 - Juego de la vida

Estas rutinas de "comunicación colectiva" han sido el tema de esta lección y nuestro programa "Juego de la vida" no es una excepción. La descripción de este problema es la misma que la utilizada en las lesssons anteriores, pero aquí utilizará una rutina de comunicación colectiva para modificar su código

## Ejercicio 6: búsqueda paralela

En este ejercicio Modifique su código de la Lección 5 para cambiar cómo el maestro envía primero los datos de destino y subcampo a los esclavos. Use las rutinas de difusión de MPI para dar a cada esclavo el objetivo. Use la rutina de dispersión MPI para dar a todos los procesadores una sección de la matriz b que buscará.

Cuando utiliza la rutina de dispersión MPI estándar, verá que la matriz global b ahora se divide en cuatro partes y el proceso maestro ahora tiene la primera cuarta parte de la matriz para buscar. Por lo tanto, debe agregar un bucle de búsqueda (similar a los esclavos) en la sección maestra del código para buscar el objetivo, calcular el promedio y luego escribir el resultado en el archivo de salida. Esto es en realidad una mejora en el rendimiento ya que todos los procesadores realizan parte de la búsqueda en paralelo.

## Autocomprobación - Comunicaciones colectivas

