Trabajo de investigación extra: Evolución y nuevas funciones de MPI

Por: Arturo Cortés Sánchez

Este trabajo cubre una pequeña introducción al estándar MPI y su diferencia con Open MPI. Un poco de historia de MPI y un subconjunto de las funciones añadidas desde el estándar 1.0 que he considerado interesantes o útiles.

1. Introducción:

Message Passing Interface (MPI) es un estándar de paso de mensajes portátil diseñado por un grupo de investigadores academicos e industriales para funcionar en una amplia variedad de arquitecturas de computación paralela. El estándar define la sintaxis y la semántica de un núcleo de rutinas de biblioteca útil para una amplia gama de usuarios que escriben programas portátiles de paso de mensajes en C, C++ y Fortran. Hay varias implementaciones de MPI bien probadas y eficientes, muchas de las cuales son de código abierto o de dominio público. Éstas fomentaron el desarrollo de una industria de software paralela y alentaron el desarrollo de aplicaciones paralelas portátiles y escalables a gran escala.

MPI es un protocolo de comunicación para la programación de ordenadores paralelos. Se apoya tanto la comunicación punto a punto como la comunicación colectiva. MPI Los objetivos de MPI son el alto rendimiento, la escalabilidad y la portabilidad. MPI sigue siendo el modelo dominante utilizado en la informática de alto rendimiento de hoy en día.

Open MPI es una implementación de código abierto y de libre acceso de las especificaciones de MPI, es utilizada por muchos superordenadores del TOP500.
Representa la fusión de tres conocidas implementaciones de MPI:

FT-MPI de la Universidad de Tennessee
LA-MPI del Laboratorio Nacional de Los Álamos
LAM/MPI de la Universidad de Indiana

Además cuenta con contribuciones del equipo PACX-MPI de la Universidad de Stuttgart. Estas cuatro instituciones son los miembros fundadores del equipo de desarrollo de Open MPI.

2. Historia del estandar MPI:

MPI-1.0:
El proceso de estandarización comenzó con el “Workshop on Standards Message-Passing in a Distributed Memory Environment”, patrocinado por el “Center for Research on Parallel Computing”, celebrado el 29 y 30 de abril de 1992 en Williamsburg, Virginia. En este evento se discutieron las características básicas esenciales para una interfaz estándar de paso de mensajes, y se estableció un grupo de trabajo para continuar el proceso de estandarización. Una propuesta de anteproyecto, conocida como MPI-1, fue presentada en noviembre de 1992, y una versión revisada fue completada en

febrero de 1993. MPI-1 trajo a la vanguardia una serie de estandarizaciones importantes pero no incluía ninguna rutina de comunicación colectiva y sus hebras no eran seguras.

MPI-1.1, MPI-1.2 y MPI-2.0:
A partir de marzo de 1995, el “MPI forum” comenzó a reunirse para considerar correcciones y la ampliación de las medidas de seguridad al documento original de MPI. El primer producto de estas deliberaciones fue la Versión 1.1 de la especificación MPI. En ese momento, el esfuerzo se centró en cinco áreas.

1. Correcciones y aclaraciones adicionales para el documento MPI-1.1.
2. Adiciones a MPI-1.1 que no cambian significativamente sus tipos de funcionalidad (nuevo constructores de tipos de datos, interoperabilidad de lenguajes, etc.).
3. Tipos de funcionalidad completamente nuevos (procesos dinámicos, comunicación unilateral, E/S paralelas, etc.) que son lo que todo el mundo considera "funcionalidad MPI-2".
4. Bindings para Fortran 90 y C++.
5. Debate en las áreas en las que el que los procesos y framework MPI pueden ser útiles, pero donde se necesita más discusión y experiencia antes de la estandarización (semánticas copia cero en maquinas de memoria compartida) .

MPI-1.3 y MPI-2.1:
Después de la publicación de MPI-2.0, el “MPI forum” siguió trabajando en las erratas y las aclaraciones para ambos documentos estándar (MPI-1.1 y MPI-2.0). El documento corto "Erratas de MPI-1.1" fue lanzado el 12 de octubre de 1998. El 5 de julio de 2001, la primera lista de erratas y aclaraciones de MPI-2.0 fue publicada, la segunda lista fue votada el 22 de mayo de 2002. Finalmente ambas listas fueron combinadas en un solo documento: "Erratas de MPI-2,".

Los trabajos de estandarizacion fueron interrumpidos unos años pero en 2008 se decide combinar los documentos MPI existentes y futuros en un solo documento para cada versión del estandar MPI. Así nacen MPI-1.3 y MPI-2.1

MPI-2.2:
MPI-2.2 es una pequeña actualización del estándar MPI-2.1. Esta versión aborda errores adicionales y ambigüedades que no se corrigieron en el estandar MPI-2.1.
El desarrollo de MPI-2.2 continuó simultáneamente con el de MPI-3; en algunos casos se propusieron extensiones para MPI-2.2, pero posteriormente se trasladaron MPI-3.

MPI-3.0:
MPI-3.0 es una actualización importante del estándar MPI. Las actualizaciones incluyen la extensión de operaciones colectivas para incluir versiones sin bloqueo, ampliaciones de las operaciones unilaterales, y nuevos bindings para Fortran 2008. Además, los bindings para C++ han sido eliminados, así como muchas de las rutinas y objetos MPI obsoletos.

MPI-3.1:
MPI-3.1 es una actualización menor al estándar MPI. La mayoría de las actualizaciones son correcciones y aclaraciones al estandar, especialmente para los bindings de Fortran. Algunas nuevas

funciones añadidas incluyen rutinas para manipular los valores de MPI_Aint de una manera portátil, rutinas colectivas de E/S no bloqueantes. También se añadió un índice general.

3. Cambios desde MPI-1.0 y nuevas funciones

El estandar MPI-1 cuenta con 129 funciones mientras que el estandar actual, el MPI-3.1 cuenta con 415. Como podemos ver, desde los años 90 se han añadido gran cantidad de funciones, aunque también se ha retirado una: MPI_Type_count, que devolvía el número de entradas de "nivel superior" en el tipo de datos.

Nuevas funciones

Versiones sin bloqueo de algunas funciones del estandar MPI-1, como por ejemplo:

- MPI_Ibcast es una versión sin bloqueo de MPI_Bcast
- MPI_Iscatter es una versión sin bloqueo de MPI_Scatter
- MPI_Ireduce es una versión sin bloqueo de MPI_Reduce
- MPI_Ibarrier es una versión sin bloqueo de MPI_Barrier
- MPI_Igather es una verriondo sin bloqueo de MPI_Gather

Funciones completamente nuevas:

MPIX_QUERY_CUDA_SUPPORT:

```
int MPIX_Query_cuda_support(void)
```

No está reconocida en el estandar MPI, es una función propia de OpenMPI devuelve verdadero si hay soporte para CUDA y falso si no lo hay.

De entrada salida:

MPI_FILE_OPEN:

```
int MPI_File_open(MPI_Comm comm, const char *filename, int amode,
MPI_Info info, MPI_File *fh)
```

MPI_File_open es una rutina colectiva que abre el archivo identificado por el nombre de archivo en todos los procesos del comunicador comm. Todos los procesos deben proporcionar el mismo valor para amode, y todos los procesos deben proporcionar nombres de archivo que hagan referencia al mismo archivo que sean textualmente idénticos. Un proceso puede abrir un archivo independientemente de otros procesos utilizando el comunicador MPI_COMM_SELF. El gestor de ficheros devuelto, fh, puede utilizarse posteriormente para acceder al fichero hasta que se cierre el fichero utilizando MPI_File_close. Antes de llamar a MPI_Finalize, el usuario debe cerrar (mediante MPI_File_close) todos los archivos que se abrieron con MPI_File_open. Hay que tener en cuenta que el comunicador comm no se ve afectado por MPI_File_open y sigue siendo utilizable en todas las rutinas MPI.

MPI_FILE_READ

```
int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype
datatype, MPI_Status *status)
```

MPI_File_read intenta leer desde el archivo asociado con fh un número total de elementos de tipo datatype en el buffer buf . Los datos se extraen de las partes del archivo especificadas en la vista actual. MPI_File_read almacena el número de elementos de tipo datatype realmente leídos en status. Todos los demás campos de status estan indefinidos.

No se puede llamar a esta función si se especificó el modo MPI_MODE_SEQUENTIAL al abrir el archivo.

MPI_FILE_WRITE

```
int MPI_File_write(MPI_File fh, const void *buf, int count, MPI_Datatype
datatype, MPI_Status *status)
```

MPI_File_write intenta escribir en el archivo asociado con fh (en la posición actual del puntero del archivo individual que mantiene el sistema) un número total de elementos de datos de recuento que tienen tipo de tipo de datos del buf del búfer del usuario. Los datos se escriben en las partes del archivo especificadas en la vista actual. MPI_File_write almacena el número de elementos de tipo de datos realmente escritos en estado. Todos los demás campos de status son indefinidos.

Es erróneo llamar a esta función si se especificó el modo MPI_MODE_SEQUENTIAL cuando se abrió el archivo.

MPI_FILE_CLOSE

```
int MPI_File_close(MPI_File *fh)
```

MPI_File_close es una rutina colectiva que primero sincroniza el estado del archivo y luego cierra el archivo asociado con fh. El usuario es responsable de asegurarse de que todas las solicitudes pendientes asociadas con fh se hayan completado antes de llamar a MPI_File_close.

Versiones no bloqueantes de las funciones de E/S

MPI_FILE_IREAD

```
int MPI_File_iread(MPI_File fh, void  *buf, int  count, MPI_Datatype
datatype, MPI_Request *request)
```

MPI_File_iread es una versión sin bloqueo de MPI_File_read. Intenta leer desde el archivo asociado con fh en la posición actual del puntero del archivo individual que mantiene el sistema, en la que un número total de elementos de datos de conteo con tipo de datos se leen en el buf del búfer del usuario. Los datos se extraen de las partes del archivo especificadas en la vista actual. MPI_File_iread almacena el número de elementos de tipo de datos realmente leídos en el estado. Todos los demás campos de

status son indefinidos. Es erróneo llamar a esta función si se especificó el modo MPI_MODE_SEQUENTIAL cuando se abrió el archivo.

MPI_FILE_IWRITE

```
int MPI_File_iwrite(MPI_File fh, const void *buf, int count,
MPI_Datatype datatype, MPI_Request *request)
```

MPI_File_iwrite es una versión sin bloqueo de MPI_File_write. Intenta escribir en el archivo asociado con fh (en la posición actual del puntero del archivo individual que mantiene el sistema) un número total de elementos de datos de conteo que tienen tipo de tipo de datos del búfer buf del usuario. Los datos se escriben en las partes del archivo especificadas en la vista actual. MPI_File_iwrite almacena el número de elementos de tipo de datos realmente escritos en estado. Todos los demás campos de estado están sin definir.

Ejemplo de E/S con OpenMPI

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
      MPI_File fh;
      int buf[1000], rank;
      MPI_Init(0, 0);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_File_open(MPI_COMM_WORLD, "test.out", MPI_MODE_CREATE |
      MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
      if (rank == 0)
            MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
      MPI_File_close(&fh);
      MPI_Finalize();
}
```

Comunicación unilateral

MPI_INFO_CREATE

```
int MPI_Info_create(MPI_Info *info)
```

MPI_Info_create crea un nuevo objeto info. El objeto recién creado no contiene pares clave/valor.

MPI_WIN_CREATE

```
MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info,
MPI_Comm comm, MPI_Win *win)
```

MPI_Win_create es una llamada colectiva de comunicación MPI unilateral ejecutada por todos los procesos comunicador comm. Devuelve un objeto ventana que puede ser utilizado por estos procesos

para realizar operaciones de acceso remoto a memoria (RMA). Cada proceso especifica una ventana de memoria existente que expone a los accesos RMA por los procesos del grupo de comm. La ventana consiste en “size” bytes, empezando por la direccion “base”. Un proceso puede elegir no exponer ninguna memoria especificando el tamaño = 0.

Si el valor base utilizado por MPI_Win_create fue asignado por MPI_Alloc_mem, el tamaño de la ventana no puede ser mayor que el valor fijado por la función MPI_ALLOC_MEM.

Se pueden usar los siguientes argumentos para “info”:

no_locks: Si se establece en true, entonces la implementación puede asumir que la ventana local nunca está bloqueada (por una llamada a MPI_Win_lock o MPI_Win_lock_all.

accumulate_ordering: Por defecto, las operaciones de acumulación de un iniciador a un objetivo en la misma ventana están estrictamente ordenadas. Si accumulate_ordering se establece en none, no se garantiza la ordenacion de las operaciones de acumulación. La clave también puede ser una lista separada por comas de los órdenes requeridos que consisten en rar, war, raw y waw (read-after-read, write-after-read, read-after-write, and write-after-write). Es probable que cuantas menos restricciones de ordenacion mayor sea el rendimiento.

accumulate_ops: Si se fija a same_op, se asumirá que todas las llamadas de acumulacion concurrentes a la misma dirección de objetivo utilizarán la misma operación. Si se establece en same_op_no_op, la implementación asumirá que todas las llamadas de acumulacion concurrentes a la misma dirección de destino utilizarán la misma operación o MPI_NO_OP. El valor por defecto es same_op_no_op.

same_size: Si se establece en true, se puede asumir que el tamaño del argumento es idéntico en todos los procesos, y que todos los procesos han proporcionado esta “info key” con el mismo valor.

same_disp_unit: Si se establece en true, entonces se puede asumir que el argumento disp_unit es idéntico en todos los procesos, y que todos los procesos han proporcionado esta “info key” con el mismo valor.

MPI_WIN_FENCE

```
int MPI_Win_fence(int assert, MPI_Win win)
```

MPI_Win_fence sincroniza las llamadas RMA en win. La llamada es colectiva en el grupo de la ventana win. Todas las operaciones de RMA que se originan en un proceso dado y se inician antes de que la llamada a MPI_WIN_FENCE se completarán antes de que la llamada a MPI_WIN_FENCE finalice. Las operaciones de RMA en win iniciadas por un proceso después de que la llamada a MPI_WIN_FENCE finalice accederán a su ventana de destino sólo después de que MPI_Win_fence haya sido llamada por el proceso de destino.

Una llamada a MPI_Win_fence suele implicar una sincronización de barreras: un proceso completa una llamada a MPI_Win_fence sólo después de que todos los demás procesos del grupo hayan hecho su llamada correspondiente. Sin embargo, una llamada a MPI_Win_fence que se sabe que no termina ningun “epoch” (en particular, una llamada con assert = MPI_MODE_NOPRECEDE) no actúa necesariamente como una barrera.

MPI_PUT

```
MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype
origin_datatype, int target_rank, MPI_Aint target_disp, int
target_count, MPI_Datatype target_datatype, MPI_Win win)
```

MPI_Put transfiere “origin_count” entradas sucesivas del tipo especificado por origin_datatype, comenzando en la direccion origin_addr en el nodo de origen al nodo de destino especificado por win y target_rank. Los datos se escriben en el búfer de destino en la dirección target_addr = window_base + target_disp x disp_unit, donde window_base y disp_unit son la dirección base y la unidad de desplazamiento de ventana especificadas en la inicialización de ventana, por el proceso de destino.

El búfer de destino se especifica mediante los argumentos target_count y target_datype.

La transferencia de datos es la misma que la que se produciría si el proceso de origen ejecutara una operación de envío con los argumentos origin_addr, origin_count, origin_datatype, target_rank, tag, comm, y el proceso de destino ejecutara una operación de recepción con los argumentos target_addr, target_count, target_datatype, source, tag, comm, donde target_addr es la dirección del búfer de destino calculada como se ha explicado anteriormente, y comm es un comunicador para el grupo de ganancia.

La comunicación debe satisfacer las mismas restricciones que para una comunicación de paso de mensajes similar. El target_datype puede no especificar entradas superpuestas en el buffer de destino. El mensaje enviado debe encajar, sin truncamiento, en el búfer de destino. Además, el búfer de destino debe encajar en la ventana de destino. Además, sólo los procesos dentro de la misma memoria intermedia pueden acceder a la ventana de destino.

MPI_GET

```
MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
origin_datatype, int target_rank, MPI_Aint target_disp, int
target_count, MPI_Datatype target_datatype, MPI_Win win)
```

MPI_Get copia los datos de la memoria de destino al origen, de forma similar a MPI_Put, excepto que la dirección de la transferencia de datos se invierte. El origin_datatype no puede especificar entradas solapadas en el buffer de origen. El búfer de destino debe estar contenido dentro de la ventana de destino y los datos copiados deben encajar, sin truncamiento, en el búfer de origen. Sólo los procesos dentro del mismo nodo pueden acceder a la ventana de destino.

Ejemplo de comunicación unilateral

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

#define NUM_ELEMENT 4
```

```c
int main(int argc, char **argv){
      int i, id, num_procs, len, localbuffer[NUM_ELEMENT],
      sharedbuffer[NUM_ELEMENT];

      char name[MPI_MAX_PROCESSOR_NAME];
      MPI_Win win;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &id);
      MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
      printf("Rank %d\n", id);
      MPI_Win_create(sharedbuffer, NUM_ELEMENT, sizeof(int), MPI_INFO_NULL,
      MPI_COMM_WORLD, &win);

      for (i = 0; i < NUM_ELEMENT; i++){
            sharedbuffer[i] = 10 * id + i;
            localbuffer[i] = 0;
      }

      printf("Rank %d pone datos en memoria compartida:", id);
      for (i = 0; i < NUM_ELEMENT; i++)
            printf(" %02d", sharedbuffer[i]);
      printf("\n");
      MPI_Win_fence(0, win);
      if (id != 0)
            MPI_Get(&localbuffer[0], NUM_ELEMENT, MPI_INT, id - 1, 0,
            NUM_ELEMENT, MPI_INT, win);
      else
            MPI_Get(&localbuffer[0], NUM_ELEMENT, MPI_INT, num_procs - 1, 0,
            NUM_ELEMENT, MPI_INT, win);

      MPI_Win_fence(0, win);
      printf("Rank %d recoge datos de memoria compartida:", id);
            for (i = 0; i < NUM_ELEMENT; i++)
                  printf(" %02d", localbuffer[i]);
      printf("\n");
      MPI_Win_fence(0, win);

      if (id < num_procs - 1)
            MPI_Put(&localbuffer[0], NUM_ELEMENT, MPI_INT, id + 1, 0,
            NUM_ELEMENT, MPI_INT, win);
      else
            MPI_Put(&localbuffer[0], NUM_ELEMENT, MPI_INT, 0, 0, NUM_ELEMENT,
            MPI_INT, win);

      MPI_Win_fence(0, win);
      printf("Rank %d tiene nuevos datos en la memoria compartida:", id);
```

```
      for (i = 0; i < NUM_ELEMENT; i++)
            printf(" %02d", sharedbuffer[i]);

      printf("\n");
      MPI_Win_free(&win);
      MPI_Finalize();
}
```

Administración dinámica de procesos

MPI_Comm_spawn

```
int MPI_Comm_spawn(const char *command, char *argv[], int maxprocs,
MPI_Info info, int root, MPI_Comm comm, MPI_Comm *intercomm, int
array_of_errcodes[])
```

MPI_Comm_spawn intenta iniciar “maxprocs” copias idénticas del programa MPI especificado por “command”, estableciendo comunicación con ellos y devolviendo un intercomunicador. Los procesos generados se denominan hijos. Los hijos tienen su propio MPI_COMM_WORLD, que está separado del del padre. MPI_Comm_spawn es una operación colectiva sobre comm, y también puede no devolver el control hasta que MPI_Init haya sido llamado en los hijos. De manera similar, MPI_Init en los niños puede no devolver el control hasta que todos los padres hayan llamado a MPI_Comm_spawn. En este sentido, MPI_Comm_spawn en los padres y MPI_Init en los hijos forman una operación colectiva sobre la unión de los procesos padre e hijo. El intercomunicador devuelto por MPI_Comm_spawn contiene los procesos padre en el grupo local y los procesos hijo en el grupo remoto. El orden de los procesos en los grupos locales y remotos es el mismo que el orden del grupo del comm en los padres y de MPI_COMM_WORLD de los hijos, respectivamente. Este intercomunicador se puede obtener en los niños a través de la función MPI_Comm_get_parent.

El estándar MPI permite que una implementación utilice el atributo MPI_UNIVERSE_SIZE de MPI_COMM_WORLD para especificar el número de procesos que estarán activos en un programa. Aunque esta implementación del estándar MPI defina MPI_UNIVERSE_SIZE, no permite al usuario fijar su valor. Si intenta establecer el valor de MPI_UNIVERSE_SIZE, obtendrá un mensaje de error. Argumentos:

- command:

  El argumento command es una cadena que contiene el nombre del programa que se va a generar. MPI busca el archivo primero en el directorio de trabajo del proceso de padre.

- argv:

  argv es un array de cadenas de caracteres que contiene argumentos que se pasan al programa. El primer elemento de argv es el primer argumento que se pasa al comando, no, como es convencional en algunos contextos, el propio comando mismo. La constante MPI_ARGV_NULL se puede utilizar para indicar una lista de argumentos vacía.

En C, el argumento MPI_Comm_spawn argv difiere del argumento argv de main en dos aspectos. Primero, se desplaza por un elemento. Específicamente, argv[0] de main contiene el nombre del programa (dado por comando). argv[1] de main corresponde a argv[0] en MPI_Comm_spawn, argv[2] de main a argv[1] de MPI_Comm_spawn, y así sucesivamente. En segundo lugar, el argv de MPI_Comm_spawn debe ser null-terminated, de modo que su longitud pueda ser determinada. Pasar un argv de MPI_ARGV_NULL a MPI_Comm_spawn da como resultado un argc de recepción principal de 1 y un argv cuyo elemento 0 es el nombre del programa.

- maxprocs:

  Open MPI intenta generar “maxprocs” procesos. Si no puede generar dichos procesos, se produce un error del tipo MPI_ERR_SPAWN. Si MPI es capaz de generar el número especificado de procesos, MPI_Comm_spawn retorna exitosamente y el número de procesos generados, m, viene dado por el tamaño del grupo remoto de intercomunicación.

  Una llamada a spawn con el comportamiento por defecto se llama "dura" y una llamada por la que se pueden devolver menos procesos que los de maxprocs se denomina “soft”.

- info:

  El argumento info es un manejador opaco del tipo MPI_Info. Es un contenedor ¡ de pares (clave, valor) especificados por el usuario, siendo clave y valor cadenas de caracteres. Las rutinas para crear y manipular el argumento info se describen en la Sección 4.10 del estándar MPI-2.

  Para las llamadas SPAWN, “info” proporciona instrucciones adicionales, dependientes de la implementación para MPI y el sistema de tiempo de ejecución sobre cómo iniciar procesos. Los programas portátiles que no requieran un control detallado sobre las ubicaciones de los procesos deben utilizar MPI_INFO_NULL.

MPI_Open_port

```
int MPI_Open_port(MPI_Info info, char *port_name)
```

MPI_Open_port establece una dirección de red, codificada en la cadena port_name, en la que el servidor podrá aceptar conexiones de clientes. port_name es suministrado por el sistema.

MPI copia el nombre de un puerto suministrado por el sistema en port_name. port_name identifica el puerto recién abierto y puede ser utilizado por un cliente para contactar con el servidor. El tamaño máximo de cadena que puede proporcionar el sistema es MPI_MAX_PORT_NAME.

MPI_Comm_connect

```
int MPI_Comm_connect(const char *port_name, MPI_Info info, int root,
MPI_Comm comm, MPI_Comm *newcomm)
```

MPI_Comm_connect establece comunicación con un servidor especificado por nombre_de_puerto. Es una operación colectiva sobre el comunicador llamante y devuelve un intercomunicador en el que el grupo remoto ha participado en el MPI_Comm_accept. La llamada MPI_Comm_connect sólo se debe llamar después de que el proceso MPI que actúa como servidor haya llamado a MPI_Comm_accept.

MPI no ofrece ninguna garantía de ordenación en los intentos de reparación de las conexiones. Es decir, los intentos de conexión no se satisfacen necesariamente en el orden en que se iniciaron, y la existencia de otros intentos de conexión puede impedir que se satisfaga un intento de conexión en particular.

El parámetro port_name es la dirección del servidor. Debe ser el mismo que el nombre devuelto por MPI_Open_port en el servidor**.**

MPI_Comm_accept

```
int MPI_Comm_accept(const char *port_name, MPI_Info info, int root,
MPI_Comm comm, MPI_Comm *newcomm)
```

MPI_Comm_accept establece la comunicación con un cliente. Es una operación colectiva sobre el comunicador llamante. Devuelve un intercomunicador que permite la comunicación con el cliente, después de que el cliente se haya conectado con la función MPI_Comm_accept usando la función MPI_Comm_connect.

El port_name debe haber sido establecido a través de una llamada a MPI_Open_port.

MPI_Comm_join

```
int MPI_Comm_join(int fd, MPI_Comm *intercomm)
```

MPI_Comm_join crea un intercomunicador a partir de la unión de dos procesos MPI que están conectados por un socket. fd es un descriptor de archivo que representa un socket del tipo SOCK_STREAM (una conexión bidireccional fiable de flujo de bytes). La E/S sin bloqueo y la notificación asíncrona a través de SIGIO no deben estar habilitadas para el socket, este debe estar en un estado conectado, y debe estar en reposo cuando se llama MPI_Comm_join.

MPI_Comm_join debe ser llamado por el proceso en cada extremo del socket. No regresa hasta que ambos procesos han llamado MPI_Comm_join.

Ejemplo de proceso dinámico

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

```c
#include <stdlib.h>

#define NUM_SPAWNS 2

int main(int argc, char *argv[]) {
      int np = NUM_SPAWNS;
      int errcodes[NUM_SPAWNS];
      MPI_Comm parentcomm, intercomm;

      MPI_Init(&argc, &argv);
      MPI_Comm_get_parent(&parentcomm);
      if (parentcomm == MPI_COMM_NULL) {
            MPI_Comm_spawn("ejecutable", MPI_ARGV_NULL, np, MPI_INFO_NULL, 0,
            MPI_COMM_WORLD, &intercomm, errcodes);
            printf("Soy el padre.\n");
      } else {
            printf("Soy el hijo.\n");
      }
      fflush(stdout);
      MPI_Finalize();
      return 0;
}
```

Referencias:

https://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps
https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
https://www.open-mpi.org/doc/v4.0/
https://www.open-mpi.org/faq/?category=general
https://en.wikipedia.org/wiki/Message_Passing_Interface
https://en.wikipedia.org/wiki/Open_MPI
https://software.intel.com/en-us/blogs/2014/08/06/one-sided-communication
http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture32.pdf
http://mpi.deino.net/mpi_functions/