UNIVERSIDAD DE BURGOS

Área de Tecnología Electrónica

Introducción a la programación en MPI.

José María Cámara Nebreda, César Represa Pérez, Pedro Luis Sánchez Ortega
Introducción a la programación en MPI. 2018
Área de Tecnología Electrónica
Departamento de Ingeniería Electromecánica
Universidad de Burgos

Introducción 1

LA PROGRAMACIÓN EN PARALELO 1
Práctica 0: Ejemplo Básico 5

OBJETIVOS 5
CONCEPTOS TEÓRICOS 5
REALIZACIÓN PRÁCTICA 7
CUESTIONES 7
DIAGRAMA DE FLUJO 8
RESULTADOS 8
Práctica 1: Comunicaciones Punto a Punto 9

OBJETIVOS 9
CONCEPTOS TEÓRICOS 9
REALIZACIÓN PRÁCTICA 12
CUESTIONES 12
DIAGRAMA DE FLUJO 13
RESULTADOS 13
Práctica 2: Comunicaciones Colectivas 15

OBJETIVOS 15
CONCEPTOS TEÓRICOS 15
REALIZACIÓN PRÁCTICA 17
CUESTIONES 17
DIAGRAMA DE FLUJO 18
RESULTADOS 19
Práctica 3: Funciones de Reparto y Reducción 21

OBJETIVOS 21
CONCEPTOS TEÓRICOS 21
REALIZACIÓN PRÁCTICA 23
CUESTIONES 23
DIAGRAMA DE FLUJO 24
RESULTADOS 25
Práctica 4: Topologías Virtuales 27

OBJETIVOS 27
CONCEPTOS TEÓRICOS 27
REALIZACIÓN PRÁCTICA 28
CUESTIONES 28
DIAGRAMA DE FLUJO 29
RESULTADOS 30
PRÁCTICA 5: PROCESOS DE ENTRADA/SALIDA 31

OBJETIVOS 31
CONCEPTOS TEÓRICOS 31

REALIZACIÓN PRÁCTICA 34
CUESTIONES 34
DIAGRAMA DE FLUJO 35
RESULTADOS 36
PRÁCTICA 6: NUEVOS MODOS DE ENVÍO 37

OBJETIVOS 37
CONCEPTOS TEÓRICOS 37
REALIZACIÓN PRÁCTICA 38
CUESTIONES 38
DIAGRAMA DE FLUJO 38
RESULTADOS 38
PRÁCTICA 7: TIPOS DE DATOS DERIVADOS 39

OBJETIVOS 39
CONCEPTOS TEÓRICOS 39
REALIZACIÓN PRÁCTICA 42
CUESTIONES 42
DIAGRAMA DE FLUJO 42
RESULTADOS 42
Práctica 8: Gestión Dinámica de Procesos 45

OBJETIVOS 45
CONCEPTOS TEÓRICOS 45
REALIZACIÓN PRÁCTICA 47
CUESTIONES 47
DIAGRAMA DE FLUJO 47
RESULTADOS 47
Práctica 9: Ejemplo de Aplicación Práctica 49

OBJETIVOS 49
CONCEPTOS TEÓRICOS 49
CUESTIONES 51
DIAGRAMA DE FLUJO 51
RESULTADOS 51
Práctica 10: Medida del rendimiento 52

OBJETIVOS 52
CONCEPTOS TEÓRICOS 52
REALIZACIÓN PRÁCTICA 56
CUESTIONES 57
Apéndice: Configuración de MS-MPI. 58

Introducción

LA PROGRAMACIÓN EN PARALELO

En este apartado introductorio vamos a contemplar las diferentes alternativas a la hora de programar sistemas que emplean paralelismo explícito, ya que aquellos que optan por el paralelismo implícito no requieren de una programación especializada.

Como sabemos, las máquinas que admiten este tipo de programación son las de **arquitectura MIMD**, tanto multiprocesadores como multicomputadores. A pesar de que nosotros vamos a trabajar con multicomputadores, haremos una breve referencia a la programación de máquinas multiprocesador.

Las arquitecturas multiprocesador hemos visto que emplean un espacio de memoria compartida por los diferentes procesadores a la que se conectan a través de un bus común con el fin de intercambiar información entre ellos para completar una labor común: el programa paralelo. El desarrollo de algoritmos adecuados para este tipo de máquinas exige partir de una abstracción del hardware sobre la que se puedan implementar algoritmos. A este efecto se crea un modelo teórico de máquina multiprocesador que no tiene reflejo en ninguna arquitectura real, pero que sirve al objetivo que se busca. Se trata de la máquina PRAM (parallel random access machine) o máquina paralela de acceso aleatorio. Este modelo presupone la existencia de una memoria compartida por todos los procesadores a la que pueden acceder en un tiempo unitario, es decir, el mismo tiempo de acceso que a su memoria local en caso de que esta exista.

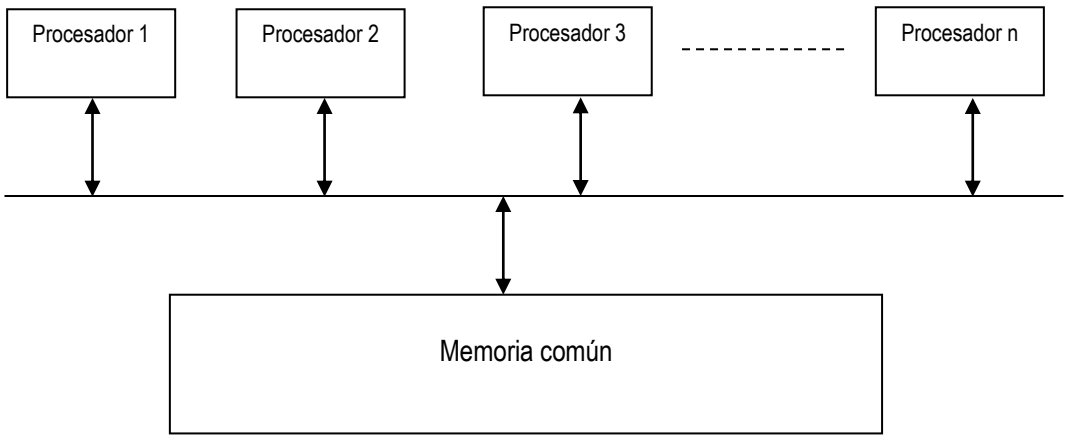


Figura 1.1. Uso de memoria común.

Este supuesto es en sí mismo imposible, pero se encuentra a medio camino entre la posibilidad real de que la memoria común sea un dispositivo independiente de la memoria local de los procesadores y la situación también real en la que la memoria común se implementa en las memorias locales de los procesadores.

Como hemos dicho, la programación de sistemas multiprocesador no es nuestro objetivo. No porque no sea interesante, que lo es y mucho, sino porque partimos de una situación diferente. Nuestro "cluster" tiene estructura de *sistema multicomputador*, independientemente de que alguno de sus nodos pueda ser un multiprocesador. En esta situación, no vamos a hacer más abstracción del

hardware que la de olvidarnos del sistema de conexión que tengamos. Lo que sí vamos a hacer es crear un modelo de programación apropiado. Consistirá en considerar los programas estructurados en procesos comunicados entre sí por canales. De esta forma, dividiremos la programación en procesos independientes, cada uno de los cuales dispone de su flujo de instrucciones secuencial, sus datos de entrada, salida e intermedios y una serie de puertos de entrada-salida que le permiten comunicarse con otros procesos a través de canales creados a tal efecto por los que se intercambian mensajes.

La mayoría de los problemas que abordemos tendrán varias posibles soluciones paralelas. Trataremos de obtener la más ventajosa, para lo cual tendremos en cuenta dos aspectos:

- Buscaremos incrementar al máximo el **rendimiento**, entendiendo éste como ejecución más rápida posible del programa completo. Para ello deberemos hacer especial hincapié en el concepto de "localidad", consistente en procurar que se realice la mayor cantidad de trabajo dentro de un proceso con datos existentes en la memoria local del computador, disminuyendo en intercambio de mensajes con otros nodos para acceder a los datos. Siempre es más rápido encontrar un dato en la propia memoria que en la de otro nodo, especialmente si está conectado a través de una red como es el caso.

- No debemos dejar de lado el concepto de **escalabilidad**, especialmente si estamos desarrollando software que habrá de correr en una arquitectura dinámica en la que el número de nodos pueda variar. En este caso, la división en procesos debe permitir aprovechar al máximo la capacidad del sistema independientemente del número de nodos disponibles en cada momento.

Vamos a explicar un posible procedimiento para desarrollar aplicaciones paralelas de forma ordenada. Debe entenderse que la programación paralela, al igual que la programación secuencial, es una tarea fundamentalmente creativa, por lo que lo que se va a exponer a continuación no pretende ser sino una secuencia de etapas que se considera interesante cubrir ordenadamente para obtener buenos resultados. El trabajo que implica cada tarea es una labor esencialmente creativa cuyos resultados dependen de las aptitudes y experiencia del programador. El procedimiento consta de cuatro etapas:

- **Fragmentación**: en esta fase inicial se intenta localizar las máximas posibilidades de paralelismo a base de descomponer la programación en tareas tan pequeñas como sea posible. No se debe plantear en esta fase la conveniencia o no de crear tareas tan simples, dado que es el punto de partida para estudiar las posibilidades de paralelización. Existen dos grandes criterios que sirven de guía para esta división:

    - El criterio funcional: contempla la naturaleza del trabajo que debe realizar el programa buscando posibles divisiones en él.

    - El criterio de datos: analiza la naturaleza de los datos que va a manejar el programa para estructurarlos en grupos de tamaño mínimo.

- **Comunicación**: partiendo de las tareas identificadas en la fase anterior, se analizan las necesidades de comunicación entre ellas.

- **Aglomeración**: dado que el coste de las comunicaciones en cuanto a rendimiento es alto, se tratará de agrupar las tareas descritas con anterioridad en otras de tamaño mayor que busquen minimizar la necesidad de comunicación. De esta forma se generarán los procesos que se van a terminar programando.

- **Mapeo**: una vez establecida la estructura del programa, falta asignar los procesos creados a los computadores disponibles. La estrategia es diferente según se haya empleado un criterio funcional o de datos a la hora realizar la fragmentación. Una condición necesaria es que existan al menos tantos procesos como máquinas, ya que de lo contrario alguna quedaría sin trabajo. Si las máquinas son iguales, sería aconsejable igualar el número de procesos al número de máquinas; si no es así, se puede asignar más procesos a las máquinas más potentes. También existe la posibilidad de que los procesos se vayan asignando de forma dinámica, analizando qué máquinas tienen una menor carga de trabajo para asignarles más. Esta situación se da especialmente en las denominadas estructuras SPMD (un solo programa con flujo de datos múltiple). En ellas todas las máquinas realizan procesos idénticos sobre datos diferentes. Cuando una de ellas termina se le asignan nuevos datos para que siga trabajando.

Siguiendo este procedimiento se puede llegar a múltiples posibilidades de programación. La experiencia orientará al programador para que pueda llegar a la más conveniente.

Práctica 0: Ejemplo Básico

OBJETIVOS

- Conocer la estructura básica de un programa paralelo.
- Aprender los conceptos básicos que se manejan en MPI a través de un ejemplo sencillo.
- Realizar la primera prueba de programación y puesta en marcha de la aplicación basada en el ejemplo anterior.

CONCEPTOS TEÓRICOS

Estructura de un programa MPI

La estructura de un programa paralelo basado en MPI es exactamente igual que la de cualquier otro programa escrito en C. Para poder utilizar las funciones MPI solamente debemos incorporar la librería de cabecera correspondiente `<mpi.h>`. Aquí se encuentran definidas las funciones que vamos a utilizar. Para poder emplearlas se ha de respetar un orden sencillo pero riguroso.

Existe un conflicto de nombres entre `stdio.h` y el `mpi.h` para C++ respecto a las funciones `SEEK_SET`, `SEEK_CUR`, y `SEEK_END`. MPI las crea en su espacio de nombres, pero `stdio.h` las define como enteros. Utilizar `#undef` puede causar problemas con otras librerías include (como `iostream.h`), por lo que se mostrará un error fatal. Para evitar este error se puede utilizar el `#undef` antes del `#include <mpi.h>`:

```
#include <stdio.h>
#undef SEEK_SET
#undef SEEK_CUR
#undef SEEK_END
#include <mpi.h>
```

o bien poner el `#include <mpi.h>` antes antes del `#include <stdio.h>` o `<iostream.h>`:

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

En la librería `<mpi.h>` se encuentran definidas las funciones que vamos a utilizar. Para poder emplearlas se ha de respetar un orden sencillo pero riguroso.

La primera función MPI será:

```
int MPI_Init(&argc, &argv[])
```

Como vemos, se le pasan como parámetros los argumentos de línea de comandos, que deberán estar declarados en la función **`main`**`( )`.

La última función MPI será:

```
int MPI_Finalize()
```

Esta función no tiene parámetros. Entre ambas se pueden emplear el resto de funciones a gusto del programador.

Un último detalle es que las implementaciones de LINUX requieren que el programa finalice con `exit(0)` o `return 0`, por lo que la función **`main`** deberá ser declarada como `int`. Esto no es necesario en la implementación para Windows, pero se mantendrá por compatibilidad.

Comunicadores

Un comunicador es una entidad virtual que representa a un número de procesos que se pueden comunicar entre sí. Supone una condición necesaria para poder establecer la comunicación, ya que dos procesos pueden intercambiarse mensajes solamente si forman parte del mismo comunicador. No obstante, cada proceso puede ser miembro de varios comunicadores.

Dentro de cada comunicador cada proceso se identifica por un número a partir de cero que se denomina rango (en inglés, '*rank*').

Un comunicador se identifica por su nombre. Por defecto, la función **`MPI_Init`** crea un comunicador del que forman parte todos los procesos denominado `MPI_COMM_WORLD`. Si se desea crear otros comunicadores habría que hacerlo explícitamente, pero no vamos a entrar en ello en este momento.

Del comunicador se puede, y normalmente se debe, obtener cierta información:

- **Rango**: en la mayoría de las aplicaciones los procesos deben conocer su propio rango de cara a identificar qué tareas deben realizar y cuales no. Pensemos, por ejemplo, que tenemos un proceso que va repartiendo cálculos a otros y se encarga de recoger los resultados. En este caso cada proceso debe saber al menos si le corresponde repartir trabajo o realizarlo. Dado que todos van a tener una copia idéntica del programa, su labor depende del rango que tengan dentro del comunicador. Generalmente, el proceso con rango cero repartiría el trabajo y el resto harían los cálculos. Para poder conocer su rango, los procesos disponen de la función:

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

  Esta función tiene como parámetro de entrada `comm` que es el nombre del comunicador. Es una variable del tipo `MPI_Comm`, tipo exclusivo de MPI. Como parámetro de salida tiene `*rank`, un puntero a un entero que sería el rango del proceso dentro del comunicador.

- **Tamaño**: suele ser interesante conocer el tamaño del comunicador, es decir, el número de procesos que lo componen. Esto sirve en muchos casos para decidir la carga de trabajo que se asigna a cada proceso. Para ello existe la función:

  ```
  int MPI_Comm_size(MPI_Comm comm, int *size)
  ```

  En este caso el parámetro de salida devuelve el dato que estábamos señalando como puntero a entero.

Otra función que puede resultar de utilidad es la que permite conocer el nombre de la máquina en la que se está ejecutando un proceso. Esta función es:

```
int MPI_Get_processor_name(char *name, int *resultlen)
```

Los dos parámetros de esta función son de salida. El primero de ellos, `name`, es una cadena de caracteres que contiene el nombre de la máquina. En el segundo parámetro, `resultlen`, la función devuelve el número de caracteres de dicha cadena.

REALIZACIÓN PRÁCTICA

En esta práctica introductoria se va a proporcionar el programa terminado para poder comprobar el comportamiento de las funciones básicas que acabamos de ver y el proceso para poner en marcha una aplicación MPI.

El programa va a ser el clásico "*¡Hola mundo!*" que se va a ejecutar en paralelo en varios procesos distribuidos en una o varias máquinas. El código fuente sería el siguiente:

```
#include <mpi.h>
#include <stdio.h>
int main (int argc,char *argv[])
{
      int mirango,tamano;
      int longitud;
      char nombre[10];
      MPI_Init (&argc,&argv);
      MPI_Comm_rank (MPI_COMM_WORLD,&mirango);
      MPI_Comm_size (MPI_COMM_WORLD,&tamano);
      MPI_Get_processor_name (nombre,&longitud);
      printf("[Maquina %s]> Proceso %d de %d: Hola Mundo!\n", nombre,
mirango, tamano);
      fflush(stdout);
      MPI_Finalize();
      return 0;
}
```

***NOTAS**:*

1. *Recordar que hay que poner* `#include <mpi.h>` *antes de* `#include <stdio.h>` *para evitar los conflictos de nombres ya explicados*.

2. *El uso de la función* **`fflush`**`(stdout)` *tiene por objeto enviar a la unidad estándar de salida (en nuestro caso, la pantalla) el resultado de la función* **`printf`**`()`*. Esto fuerza a mostrar los resultados por pantalla antes de seguir con la ejecución del programa*.

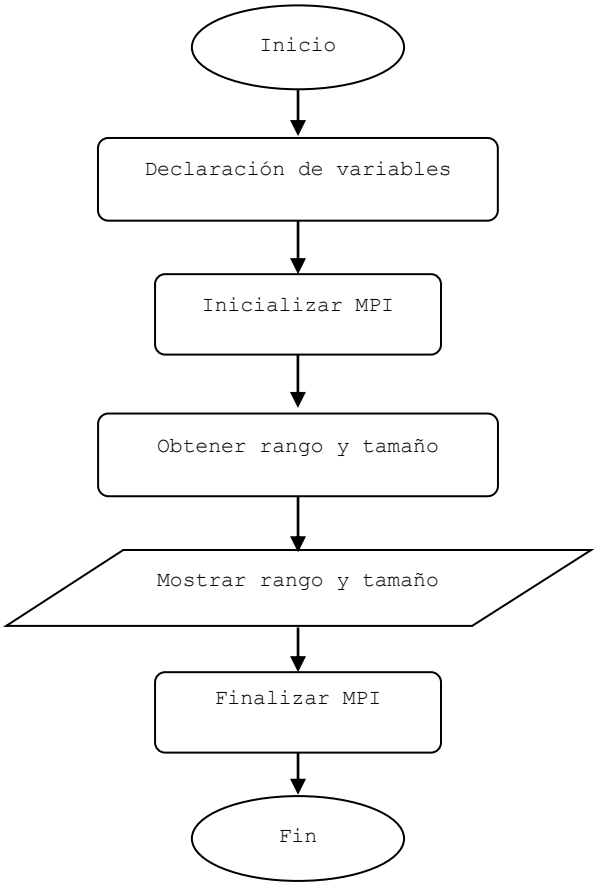
Para poner en marcha esta aplicación es necesario como siempre, teclearla, construirla y ejecutarla. Para los dos primeros pasos se va a emplear el entorno de trabajo de Microsoft Visual Studio. La configuración de un proyecto en Visual C++ se describe con detalle en el apéndice B de este manual.

Para ejecutar la aplicación será necesario lanzarla desde el entorno de DeinoMPI que se encuentra instalado como aplicación en todas las máquinas que forman el cluster. En el apéndice A del manual podemos ver los detalles de manejo de esta aplicación.

CUESTIONES

- ¿Es posible entrever, a la vista de los resultados, el criterio que emplea el sistema para asignar rango a los procesos dentro de un comunicador?
- Explicar la reacción del sistema cuando no se inicia o termina MPI o se insertan funciones MPI fuera del espacio comprendido entre estos dos eventos.

DIAGRAMA DE FLUJO



RESULTADOS

[Maquina Sony-VAIO]> Proceso 2 de 4: Hola Mundo!
[Maquina Sony-VAIO]> Proceso 0 de 4: Hola Mundo!
[Maquina Sony-VAIO]> Proceso 1 de 4: Hola Mundo!
[Maquina Sony-VAIO]> Proceso 3 de 4: Hola Mundo!

Práctica 1: Comunicaciones Punto a Punto

OBJETIVOS

- Estudiar los diferentes modos de comunicación entre procesos y la estructura de los mensajes intercambiados.
- Conocer las funciones de paso de mensajes básicas proporcionadas por MPI.
- Programar una primera aplicación de reparto de trabajo basada en paso de mensajes.

CONCEPTOS TEÓRICOS

Comunicación entre procesos

Podemos estudiar y por lo tanto clasificar los modos de comunicación entre procesos en base a diferentes criterios. En este caso, los criterios no son alternativos, sino complementarios, por lo que debemos contemplarlos todos para tener una visión completa de las posibilidades de comunicación:

- Según el número de procesos que generan y reciben la información tenemos comunicaciones:
  - *Punto a punto*: un proceso envía información a otro proceso.
  - *Punto a multipunto (broadcast)*: un proceso envía información a varios.
  - *Multipunto a punto*: varios procesos envían información a uno. Esto es físicamente inviable porque produciría colisiones en el canal de comunicación (red local), pero MPI nos proporciona una función que virtualiza esta posibilidad.
- En función de la sincronización entre emisor y receptor podemos tener:
  - *Envíos síncronos*: el proceso emisor queda bloqueado hasta que el receptor recoge el mensaje.
  - *Envíos asíncronos*: el proceso emisor copia el mensaje en un buffer interno y lo envía en “background”.
- Según el bloqueo de los procesos, puede ocurrir:
  - Que los procesos emisor y receptor se bloqueen hasta que las correspondientes operaciones se hayan completado. Para ello no es necesario en el caso del emisor que el receptor esté dispuesto, sino que en caso de disponer de buffer interno, se considera la operación completada cuando los datos han sido transferidos a dicho buffer. Sólo quedaría parado el proceso cuando el buffer esté lleno.

- Que los procesos no se bloqueen y sigan corriendo aún cuando las operaciones de envío o recepción no se hayan completado. Esto no acarrea mayores problemas siempre que se tenga en cuenta para no sobrescribir datos que aún no han sido enviados o para no recoger datos antiguos según el caso.

Mensajes

Un mensaje MPI consta de dos partes: envoltura y cuerpo.

La envoltura está compuesta de:

- **Fuente**: Identificación (rango) del emisor.
- **Destino**: Identificación (rango) del receptor.
- **Comunicador**: Nombre del comunicador al que pertenecen fuente y destino.
- **Etiqueta**: Número que emplean emisor y receptor para clasificar los mensajes.

El cuerpo está formado por:

- **Buffer**: Zona de memoria en la que reside la información de salida (para envío) o de entrada (para recepción).
- **Tipo de datos**: Pueden ser datos simples: `int`, `float`, etc; o datos complejos definidos por el usuario.
- **Cuenta**: Número de datos del tipo definido que componen el mensaje.

En lo que se refiere a los datos, MPI dispone de sus propios tipos de datos estándar, de manera que los hace independientes de la máquina o máquinas con las que estemos trabajando. De este modo no es necesario ocuparse de aspectos tales como el formato en punto flotante empleado por cada una.

Funciones de envío y recepción.

MPI permite implementar diferentes modos de envío en consonancia con lo que se ha comentado anteriormente. No obstante permite algunas variantes adicionales como el modo preparado y el modo con buffer en las que de momento no vamos a entrar. Nos vamos a quedar con lo que MPI considera el modo de envío estándar. Éste puede ser síncrono o asíncrono según decida MPI en función de los recursos disponibles. Dentro de este modo vamos a ver solamente las funciones de envío y recepción bloqueante. Éstas son:

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

int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int source, int
   tag, MPI_Comm comm, MPI_Status *status)
```

La función de envío **`MPI_Send`** dispone de los siguientes argumentos todos ellos de entrada:

- `*buf`: Puntero al buffer en que se encuentran los datos.

- `count`: Número de datos del mensaje.
- `dtype`: Formato de los datos.
- `dest`: Rango del destinatario dentro del comunicador.
- `tag`: Etiqueta del mensaje.
- `comm`: Comunicador al que pertenecen fuente y destino.

La función de recepción **`MPI_Recv`** dispone de los siguientes argumentos de entrada:

- `count`: Número de datos del mensaje.
- `dtype`: Formato de los datos.
- `source`: Rango del emisor dentro del comunicador.
- `tag`: Etiqueta del mensaje.
- `comm`: Comunicador al que pertenecen fuente y destino.

Los argumentos de salida son:

- `*buf`: Puntero al buffer en que se encuentran los datos.
- `*status`: Devuelve datos sobre el mensaje tales como la fuente y la etiqueta. Se declara como variable tipo `MPI_Status`.

Ambas funciones devuelven un código de error en caso de no haber podido completar su ejecución. Estas funciones trabajan en modo bloqueante.

Vamos a proporcionar a modo de referencia rápida una tabla con los tipos de datos manejados por MPI y su equivalencia en C:

Tipo MPI	Tipo C
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long double
MPI_BYTE	Ninguno
MPI_PACKED	Ninguno

Tabla 1. Tipos de datos manejados por MPI.

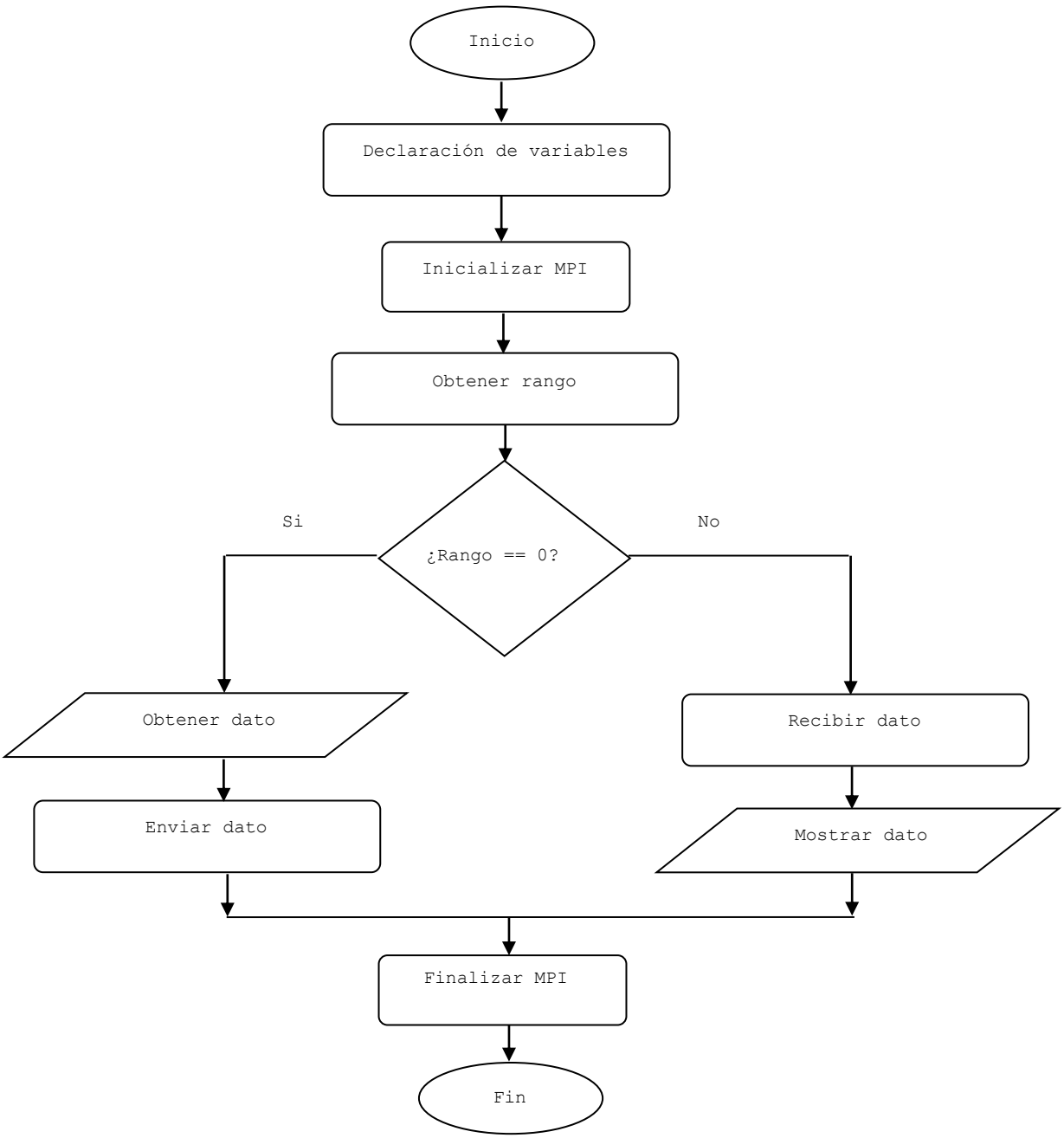
REALIZACIÓN PRÁCTICA

Vamos a realizar una sencilla aplicación de envío y recepción empleando el modo estándar bloqueante cuyas funciones ya se conocen. Se trata de que el proceso de rango 0 le envíe un dato numérico introducido por el usuario al proceso de rango 1 que lo visualizará.

CUESTIONES

- ¿Cómo se podría enviar el mismo dato a varios procesos?
- ¿Qué orden de recepción se prevé que existiría: por rango, por proximidad geográfica, por orden de programa...?

DIAGRAMA DE FLUJO



RESULTADOS

```
Soy el proceso 2 de 2: Hola Mundo!
Voy a recibir el numero de otro proceso...
Soy el proceso 1 de 2: Hola Mundo!
Introduce un numero entero:
5
El numero introducido es n=5
Voy a enviar el numero a otro proceso...
Dato enviado!
Me ha llegado el numero n=5!!
```

Práctica 2: Comunicaciones Colectivas

OBJETIVOS

- Extender las posibilidades de comunicación entre procesos a las comunicaciones colectivas como método de simplificación de la comunicación.
- Estudiar posibles aplicaciones de las comunicaciones colectivas.

CONCEPTOS TEÓRICOS

Comunicaciones colectivas

En la práctica anterior estudiamos la manera básica de pasar mensajes entre procesos. A pesar de que parece un procedimiento de cierta complejidad que se concreta en diferentes posibilidades de diálogo entre los procesos, no resulta en muchos casos, la opción más adecuada. En este caso, complejidad no es sinónimo de potencia. En la mayoría de las aplicaciones prácticas del paso de mensajes, el escenario consiste en un proceso (maestro) que reparte datos a muchos (esclavos) y que recopila los resultados que éstos le proporcionan. Es posible manejar el tráfico de mensajes que esto genera empleando las funciones de envío y recepción que se emplearon en la práctica anterior, pero si se dispone de funciones más potentes, enseguida se descartará esta posibilidad.

Las funciones de comunicación colectiva permiten que un proceso envíe datos a varios y que recoja resultados de otros tantos de una sola vez. Esto simplifica la programación de la inmensa mayoría de las aplicaciones.

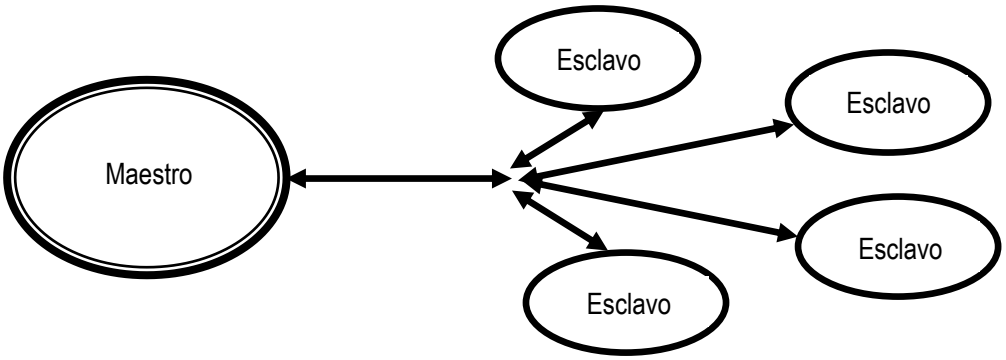


Figura 2.1. Estructura Maestro-Esclavo.

Implementación en MPI

El concepto de *broadcast* se asocia tradicionalmente a envíos programados desde un emisor a todos los posibles receptores. Se puede traducir a castellano como difusión. MPI proporciona una función para poder realizar este tipo de envíos en una única llamada. Se trata de la función:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype dtype, int source,
   MPI_Comm comm)
```

El significado de los parámetros ya lo conocemos. Simplemente cabe observar que es un envío con fuente pero sin destino. Esto quiere decir que esta función se emplea por igual en el código de todos los procesos, pero solamente aquel cuyo rango coincida con `source` va a enviar, el resto entiende

que deben recoger lo que aquel envíe. En la figura 2.2 tenemos el esquema de trabajo de la función **`MPI_Bcast`** donde se puede apreciar que esta función copia el contenido del buffer `*buf` del proceso `source` y lo copia en el resto de procesos.

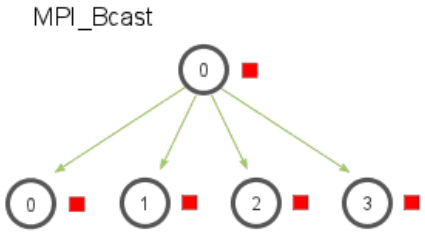


Figura 2.2. Esquema de trabajo de la función **`MPI_Bcast`**.

El concepto contrario a la difusión lo podemos denominar *recolección*. Consiste en recopilar la información generada en forma de resultados por los procesos esclavos. MPI proporciona la posibilidad de concentrar todos los resultados en el proceso maestro o en todos los procesos. Las funciones que realizan estas operaciones son respectivamente:

```
int MPI_Gather(void *bufsource, int count, MPI_Datatype dtype, void
   *bufdest, int count, MPI_Datatype dtype, int dest, MPI_Comm comm)

int MPI_Allgather(void *bufsource, int count, MPI_Datatype dtype, void
   *bufdest, int count, MPI_Datatype dtype, MPI_Comm comm)
```

En el primer caso, los datos proporcionados por todos los procesos (`*bufsource`) se almacenan en el buffer de aquel cuyo rango coincida con el valor `dest` (`*bufdest`). En el segundo caso los datos se copian en los buffer de todos los procesos. Por este motivo no existe el argumento de destino `dest`. Los datos quedan automáticamente ordenados por orden de rango del emisor en ambos casos.

El valor de `count` es el mismo para origen y destino curiosamente. Se trata de la cantidad de datos que se envían y de los que se reciben de cada proceso respectivamente.

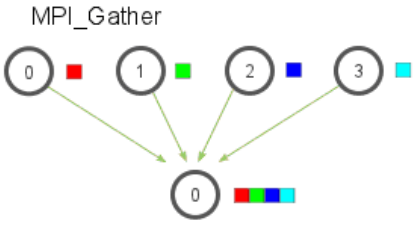


Figura 2.3. Esquema de trabajo de la función **`MPI_Gather`**.

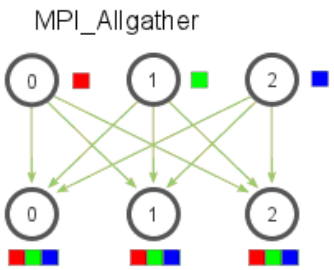


Figura 2.4. Esquema de trabajo de la función **`MPI_Allgather`**.

REALIZACIÓN PRÁCTICA

Como aplicación de los conceptos estudiados vamos a desarrollar un programa de suma de matrices de tamaño N×N. El proceso 0 se encargará de generar el contenido de dos matrices y distribuirlo al resto de procesos. Cada uno de ellos sumará una columna y el proceso cero recopilará todos los resultados para mostrarlos en pantalla. Para que el proceso sea correcto, el número de procesos lanzados tendrá que ser igual al rango de las matrices.

Vamos a añadir la posibilidad de medida del rendimiento de la aplicación. Para ello, MPI proporciona un reloj que nos permite medir la duración del cálculo. Se implementa mediante la función:

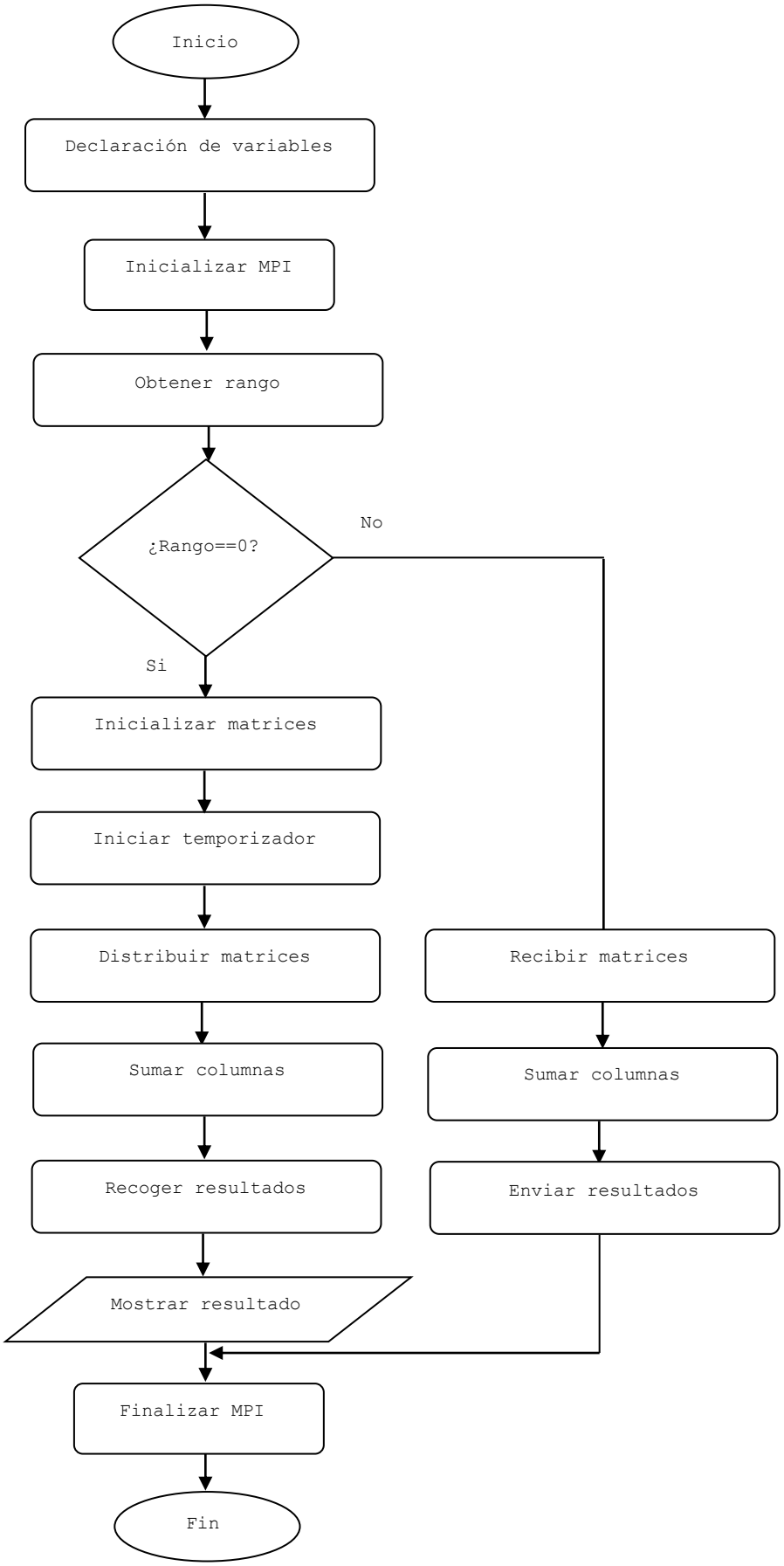
```
double MPI_Wtime(void)
```

que proporciona el tiempo en el instante actual en segundos.

CUESTIONES

- Las comunicaciones colectivas facilitan la programación y simplifican el código. ¿Se puede pensar que acortan el tiempo de ejecución de los programas?
- Explicar qué refleja la medida de tiempo realizada.
- Plantear otras posibilidades de medida de tiempos de ejecución que permitan distinguir los tiempos invertidos en comunicación entre procesos y los tiempos dedicados al cálculo.

DIAGRAMA DE FLUJO



RESULTADOS

```
El numero de procesos lanzados es Size=10
El tamano de la matriz es 10x10
El tiempo empleado ha sido de 0.012847 segundos
100 101 102 103 104 105 106 107 108 109
110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129
130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149
150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169
170 171 172 173 174 175 176 177 178 179
180 181 182 183 184 185 186 187 188 189
190 191 192 193 194 195 196 197 198 199
```

Práctica 3: Funciones de Reparto y Reducción

OBJETIVOS

- Estudiar opciones más potentes dentro de las comunicaciones colectivas.

CONCEPTOS TEÓRICOS

Operaciones de reparto y reducción

Vistas las funciones básicas de comunicación colectiva comprenderemos las notables ventajas que ofrecen respecto de las comunicaciones punto a punto. No obstante, es posible incrementar la potencia de la comunicación añadiendo alguna funcionalidad extra. Con esta misión se crean las funciones de reparto y reducción. Una operación de reparto consiste en la distribución de un conjunto de datos entre una colección de procesos. A cada proceso le corresponde un subconjunto de los datos totales. La reducción consiste en añadir a la recolección que vimos en los apartados anteriores una operación a realizar sobre los datos entrantes, generando con ellos un resultado final.

Implementación en MPI

MPI proporciona sendas funciones dentro de su librería para implementar operaciones de reparto y reducción.

La función de reparto es:

```
int MPI_Scatter(void *bufsend, int count, MPI_Datatype dtype, void
   *bufrecv, int count, MPI_Datatype dtype, int source, MPI_Comm comm)
```

Como vemos, su funcionamiento es bastante análogo a funciones anteriormente vistas. En este caso, parte de los datos contenidos en el buffer del proceso emisor (`*bufsend`) son copiados en los buffers de los procesos receptores (`*bufrecv`). El primer parámetro `count` indica cuántos datos van a cada proceso. Existe una pequeña diferencia entre las funciones **`MPI_Bcast`** y **`MPI_Scatter`** pero importante. Mientras que la función **`MPI_Bcast`** envía el mismo conjunto de datos a todos los procesos, la función **`MPI_Scatter`** envía únicamente trozos de dicho conjunto. Como podemos apreciar en la figura 3.1, la función **`MPI_Scatter`** toma un array de elementos y los reparte entre todos los procesos por orden de rango. En la figura 3.1, el primer elemento (en rojo) va al proceso de rango cero, el segundo elemento (verde) va al proceso de rango 1, y así sucesivamente. Es importante notar que aunque el proceso cero contiene el array completo, la función **`MPI_Scatter`** copiará el elemento apropiado en el buffer de todos los procesos receptores, incluido él mismo.

Figura 3.1. Diferencia entre las funciones **`MPI_Bcast`** *y* **`MPI_Scatter`**.

La función de reducción es:

```
int MPI_Reduce(void *bufsend, void *bufrecv, int count, MPI_Datatype
   dtype, MPI_op operation, int dest, MPI_Comm comm)
```

De manera parecida a la función **MPI_Gather**, esta función permite recopilar la información generada por los procesos esclavos. Así, ante esta función, cada proceso envía `count` datos almacenados en el buffer `*bufsend` a aquel proceso cuyo rango coincide con el parámetro `dest`, el cual no los almacena directamente en su buffer `*bufrecv`, sino que lo que ahí coloca es el resultado de realizar sobre estos datos la operación indicada en `operation` (figura 3.2). Este parámetro es del tipo `MPI_op`, es decir, una operación MPI. Las operaciones MPI más habituales aparecen en la Tabla 2.

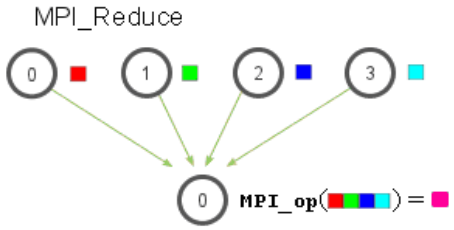


Figura 3.2. Operación de reducción realizada con la función **MPI_Reduce**.

Operación	Descripción
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Suma
MPI_PROD	Producto
MPI_LAND	Y lógico
MPI_LOR	O lógico
MPI_LXOR	XOR lógico
MPI_BXOR	XOR a nivel de bits
MPI_MINLOC	determina el rango del proceso que contiene el valor menor.
MPI_MAXLOC	determina e rango del proceso que contiene el valor mayor.

Tabla 2. Operaciones MPI.

REALIZACIÓN PRÁCTICA

En esta práctica se va a programar el producto escalar entre dos vectores de tamaño cualesquiera:

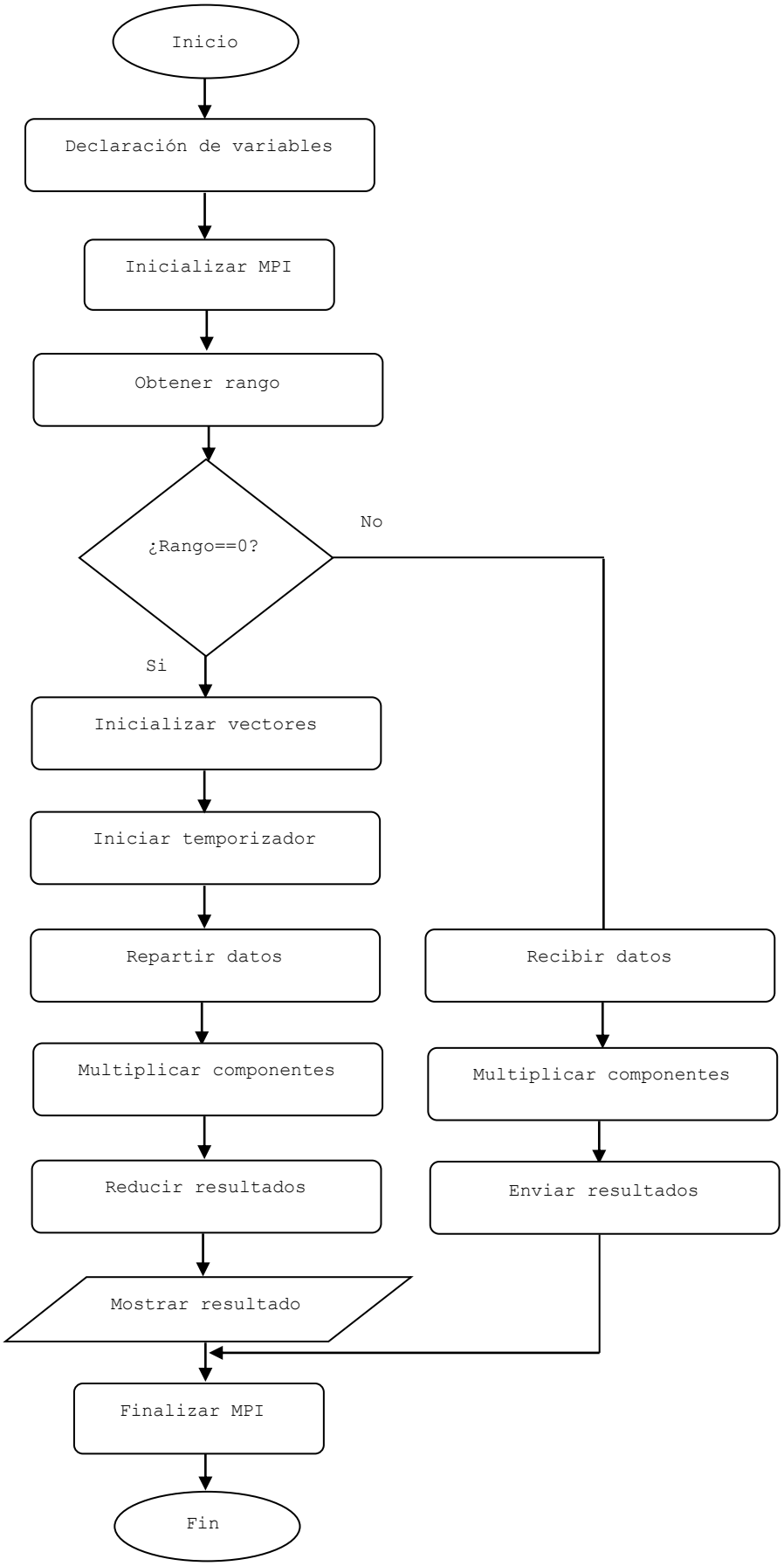
$$(x_0, x_1, ..., x_n)\cdot (y_0, y_1, ..., y_n) = x_0 \cdot y_0 + x_1 \cdot y_1 + ... + x_n \cdot y_n$$

Se propone construir la aplicación de manera que cada proceso calcule un término de la suma, por lo que el número de procesos lanzados debe coincidir con el tamaño de los vectores. El proceso 0 se encargará de inicializar los arrays correspondientes a los vectores **x** e **y** y de repartir un elemento a cada proceso. Finalmente el proceso 0 recogerá las multiplicaciones parciales y mostrará el resultado del producto escalar, junto con el tiempo empleado.

CUESTIONES

- Plantear la posibilidad de optimizar el funcionamiento del sistema teniendo en cuenta que los procesos de rango más alto realizan más trabajo que los de rangos bajos, al ser sus operaciones más complejas. Esto se podría aprovechar para lanzar más cálculos a los procesos de rangos bajos.
- En la situación anterior ¿se puede mantener la utilidad de la operación de reducción?

DIAGRAMA DE FLUJO



RESULTADOS

```
P0: Vector 1:
0 1 2 3
P0: Vector 2:
0 1 2 3
P0: El resultado es: 14
```

Práctica 4: Topologías Virtuales

OBJETIVOS

- Realizar un acercamiento a la configuración de topologías virtuales como herramienta para potenciar la resolución de determinados problemas.

CONCEPTOS TEÓRICOS

Topología cartesiana en MPI

Hasta ahora hemos visto que cualquier conjunto de procesos que quiera intercambiar información debe pertenecer a un mismo comunicador. Como la complejidad de los programas realizados hasta este momento ha sido baja, solamente se ha empleado el comunicador por defecto `MPI_COMM_WORLD`. Esta situación no se va a modificar. Lo que vamos a ver en esta práctica es la posibilidad de dotar a los procesos miembros de un comunicador de una distribución virtual análoga a la del problema que va a resolver.

Nos vamos a centrar en la topología cartesiana. Consiste en la distribución matricial de los procesos. Para ello se define el número de dimensiones de la matriz de procesos y posteriormente, a cada proceso se le asignan coordenadas que identifican su posición dentro de la matriz. Los procesos siguen perteneciendo a un comunicador y disponen de su rango dentro de él, pero disponen de una nueva referencia más útil dentro del programa que se va a desarrollar.

Existen varias funciones asociadas a la creación y manejo de topologías virtuales. Vamos a ver las principales:

```
int MPI_Cart_create(MPI_Comm comm1, int ndims, int *dim_size, int
   *periods, int reorder, MPI_Comm *comm2)
```

Esta función renombra los procesos incluidos en `comm1` (MPI_COMM_WORLD) empleando para ello coordenadas cartesianas incluyéndolos en el nuevo comunicador apuntado por `*comm2`. Crea una topología formada por una matriz de procesos de `ndims` dimensiones. El tamaño de cada dimensión viene dado por `*dim_size`, que resulta ser un puntero a un vector con tantos elementos como dimensiones. Cada elemento corresponde con el tamaño de una dimensión. Así, si se pretende que 12 procesos formen una matriz bidimensional de 4 filas y 3 columnas tendremos que inicializar `ndims` a 2 y crear un vector de dos enteros, donde el primero de los cuales valdrá 4 y el segundo 3.

Un significado un poco más complejo tienen los parámetros `*periods` y `reorder`. El primero de ellos (`*periods`) es un puntero a un vector de dimensión `ndims` que indica en cada componente si en esa dimensión la matriz que se está creando es periódica (**1**) o no (**0**). El sentido que esto tiene es el siguiente: supongamos que se está creando una estructura cartesiana con M filas y N columnas. Esto asignaría coordenadas a M×N procesos. En caso, por ejemplo, de declarar periódicas las filas, cualquier referencia a la fila M, se asimilaría de nuevo a la fila 0; una referencia la fila M+1 se entendería como una referencia a la fila 1 y así sucesivamente. Si las filas o columnas se declaran no periódicas, cualquier referencia fuera del rango existente genera una situación de error.

El segundo parámetro (`reorder`) autoriza a MPI a modificar (**`1`**) el orden de los procesos en el nuevo comunicador respecto al que tenían en el antiguo. No es algo que deba preocupar en este momento.

Para poder asignar trabajo a los procesos en función de sus coordenadas, deberemos conocer cuáles son estas. Deberá existir una función equivalente a **`MPI_Comm_rank`** que en este caso nos devuelva las coordenadas cartesianas del actual proceso. Esta función es:

```
int MPI_Cart_coords(MPI_Comm comm2, int rank, int ndims, int *coords)
```

Esta función devuelve en el vector apuntado por `*coords` (cuya dimensión debe coincidir con `ndims`) las coordenadas cartesianas dentro del nuevo comunicador `comm2` del proceso cuyo rango es `rank`.

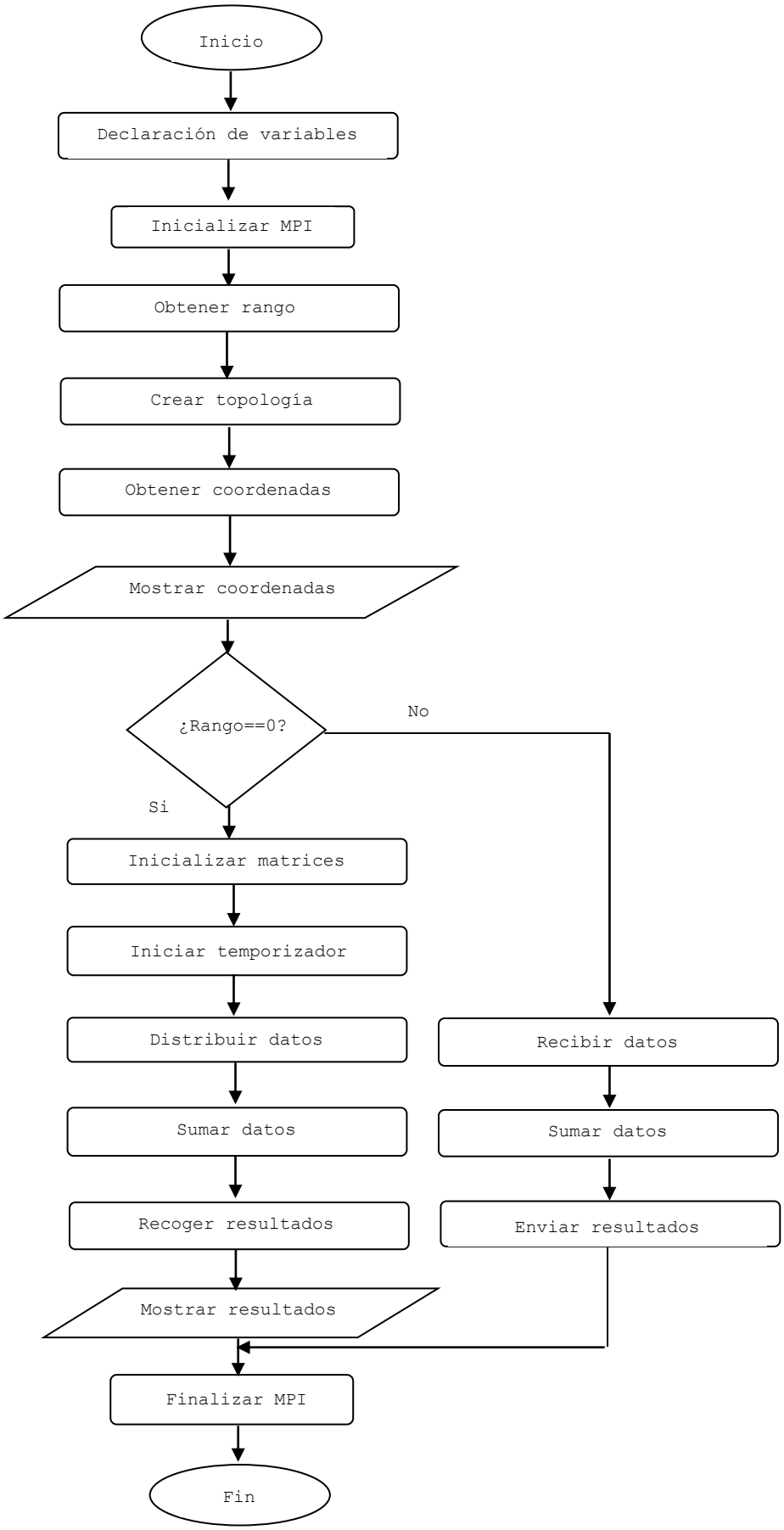
REALIZACIÓN PRÁCTICA

Se reprogramará la suma de matrices N×N de manera que cada proceso dentro de una topología cartesiana sume uno de los elementos de cada matriz. El resultado se mostrará, junto con el tiempo empleado, de igual manera que en la práctica 2.

CUESTIONES

- ¿Cuál es la distribución en memoria de los datos contenidos dentro de matrices en C?
- Pensar posibles aplicaciones en las que se pueda aprovechar la topología cartesiana.
- Plantear otras posibles topologías que puedan aplicarse en diferentes situaciones.

DIAGRAMA DE FLUJO



RESULTADOS

```
Proceso 0 de 15: Version MPI 2.0
El numero de procesos lanzados es Size=15
El tamano de la matriz es 3x5
Soy el proceso  0 y ahora soy [0][0]
El tiempo empleado ha sido de 0.039157 segundos
100 101 102 103 104
110 111 112 113 114
120 121 122 123 124
Soy el proceso 12 y ahora soy [2][2]
Soy el proceso  8 y ahora soy [1][3]
Soy el proceso  4 y ahora soy [0][4]
Soy el proceso 13 y ahora soy [2][3]
Soy el proceso  9 y ahora soy [1][4]
Soy el proceso  5 y ahora soy [1][0]
Soy el proceso  1 y ahora soy [0][1]
Soy el proceso  6 y ahora soy [1][1]
Soy el proceso  2 y ahora soy [0][2]
Soy el proceso 14 y ahora soy [2][4]
Soy el proceso 10 y ahora soy [2][0]
Soy el proceso  7 y ahora soy [1][2]
Soy el proceso  3 y ahora soy [0][3]
Soy el proceso 11 y ahora soy [2][1]
```

PRÁCTICA 5: PROCESOS DE ENTRADA/SALIDA

OBJETIVOS

- Introducirse en las técnicas de entrada/salida en paralelo.

CONCEPTOS TEÓRICOS

Entrada/Salida serie

En las aplicaciones tradicionales existen procesos de entrada/salida. Estos procesos consisten en la lectura de datos previos a la computación y la escritura de resultados posteriores a ella. Estas operaciones las realiza el único proceso en marcha. En las aplicaciones paralelas que manejamos se puede mantener esta forma de trabajo sin ningún problema. De hecho, en las prácticas anteriores ha sido así. Hasta ahora, ha habido un proceso (el proceso 0) que se encargaba, si era el caso, de recoger los datos de partida, repartirlos al resto de procesos y mostrar los resultados generados por la máquina paralela. Esto tiene el problema de que se genera un potencial cuello de botella en el proceso 0 ya que debe comunicarse con todos los demás. Si cada uno de los procesos fuera capaz de leer sus datos y/o escribir sus resultados, se evitaría este cuello de botella. Esto es lo que se denomina entrada/salida en paralelo; la estrategia anterior era entrada/salida serie.

Entrada/salida en paralelo

Sobre la idea anterior, la entrada/salida en paralelo exige que todos los procesos puedan acceder a un mismo fichero para realizar en él las operaciones habituales: lectura y/o escritura. Esto resultará sin duda de gran utilidad, pero para que funcione correctamente deberán aclararse algunas cuestiones.

En primer lugar, se debe establecer cómo ve cada proceso el fichero común. Es evidente que los procesos no pueden leer y escribir arbitrariamente datos en el fichero ya que en tal caso sobrescribirían datos de otros procesos o leerían cosas absurdas. La situación habitual será, por tanto, que cada proceso tenga su propia zona de fichero para sus datos, es decir, tendrá una vista propia del fichero. Esto no es óbice para que eventualmente un proceso pueda acceder para lectura o escritura a la zona de datos de otro, pero de una forma ordenada.

Otra cuestión es la forma de operar sobre el estado del fichero: crearlo, abrirlo para lectura, para escritura, cerrarlo, etc. En este sentido, cada proceso deberá poder trabajar como si fuera él el único que maneja el fichero.

Entrada/salida en paralelo en MPI

MPI proporciona una serie de funciones que permiten realizar las operaciones mencionadas en el apartado anterior de una forma sencilla. La primera de ellas es la que permite abrir un fichero:

```
int MPI_File_open(MPI_Comm com, char *fichero, int modo, MPI_Info info,
   MPI_File *fh)
```

Esta función permite a un proceso perteneciente al comunicador especificado abrir un fichero cuyo nombre viene dado por `*fichero` para realizar en él la operación definida en `modo`. La función devuelve un puntero (`*fh`) que sirve como manejador del fichero de cara a su empleo en otras funciones. El parámetro `info` hace referencia a un manejador de información del proceso cuyo significado varía con cada implementación de MPI. Se suele pasar como nulo (`MPI_INFO_NULL`) dada su escasa utilidad. Finalmente se debe especificar que los modos de acceso al fichero pueden ser los siguientes:

Modo de acceso	Descripción
MPI_MODE_RDONLY	Sólo lectura
MPI_MODE_WRONLY	Sólo escritura
MPI_MODE_RDWR	Lectura y escritura
MPI_MODE_CREATE	Crear fichero si no existe
MPI_MODE_EXCL	Devolver error si se crea un fichero que ya existe
MPI_MODE_DELETE_ON_CLOSE	Borrar el fichero cuando se cierre
MPI_MODE_UNIQUE_OPEN	No se permiten accesos concurrentes al fichero
MPI_MODE_SECUENTIAL	Sólo puede ser accedido secuencialmente
MPI_MODE_APPEND	La posición inicial de todos los punteros se establece al final del fichero

Tabla 3. Modos de acceso a fichero en MPI.

Es lógico que en muchos casos se necesite establecer varias de estas condiciones al abrir un fichero. No hay ningún problema en hacerlo. Por ejemplo, una forma habitual de abrir un fichero sería crearlo en caso de que no exista y abrirlo para lectura y escritura. Esto se puede hacer sintácticamente mediante la siguiente construcción:

```
MPI_MODE_CREATE | MPI_MODE_RDWR
```

La siguiente operación que suele aparecer cuando se trata con ficheros compartidos es la definición de la vista que cada proceso va a tener del fichero. Esto consiste en especificarle a cada uno dónde empieza su zona del fichero, qué estructura tiene y qué tipo de datos va a albergar. Para ello se dispone de la siguiente función:

```
int MPI_File_set_view(MPI_File fh, MPI_Offset despl, MPI_Datatype
   dtipo, MPI_Datatype ftipo, char *datarep, MPI_Info info)
```

Emplea el manejador proporcionado por la función anterior para hacer referencia al fichero. En él define una zona a partir de la posición `despl` en la cual se van a almacenar datos del tipo `dtipo`. La estructura del fichero definida por `ftipo` establece la forma en que se van a intercalar los datos introducidos por los diferentes procesos. Este parámetro proporciona una notable potencia a la entrada/salida en paralelo ya que permite intercalar datos de diferentes tipos y tamaños proporcionados por diferentes procesos. Para ello es necesario construir un tipo derivado de datos

MPI que tenga la estructura que se pretende que se repita dentro del fichero. Por el momento no vamos a aprovechar esta funcionalidad y simplemente declararemos `ftipo` igual que `dtipo`. De esta manera nuestros ficheros de datos tendrán una estructura homogénea. La representación de los datos `*datarep` especifica la forma de representar los datos en el fichero al trasladarlos a él desde memoria. Puede tener tres valores: "`native`", "`internal`" o "`external32`". Los formatos "`native`", son propios de cada implementación de MPI. En este formato los datos se trasladan de memoria al fichero sin ninguna modificación. Esto es óptimo de cara a que no se pierde tiempo en conversiones de datos y no existe pérdida de precisión, pero no permite que máquinas que corran diferentes implementaciones de MPI compartan estos datos. La representación "`external32`" es entendida por cualquier implementación de MPI. Su inconveniente es que exige siempre una conversión de formatos. El caso intermedio es la representación "`internal`" que solamente realiza la conversión de datos si es necesaria para que estos sean entendidos por todas las máquinas. Utilizaremos la representación en nuestro "`native`" caso.

Tras declarar la vista ya se está dispuesto a leer o escribir en el fichero. Existen varias formas de lectura y escritura en MPI proporcionadas por otras tantas funciones. Vamos a ver el caso más simple:

```
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int
   count, MPI_Datatype dtipo, MPI_Status *estado)
```

Esta función permite leer el fichero `fh` a partir de la posición `offset` e introducir los datos encontrados en él hasta el número `count` en el vector `*buf`. Los datos serán del tipo `dtipo`.

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf, int
   count, MPI_Datatype dtipo, MPI_Status *estado)
```

Esta función permite escribir el fichero `fh` a partir de la posición `offset` e introducir en él los datos encontrados en el vector `*buf` hasta el número `count`. Los datos serán del tipo `dtipo`.

Finalmente se deberá cerrar el fichero, toda vez que no se vaya a volver a utilizar dentro del programa. Para ello:

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

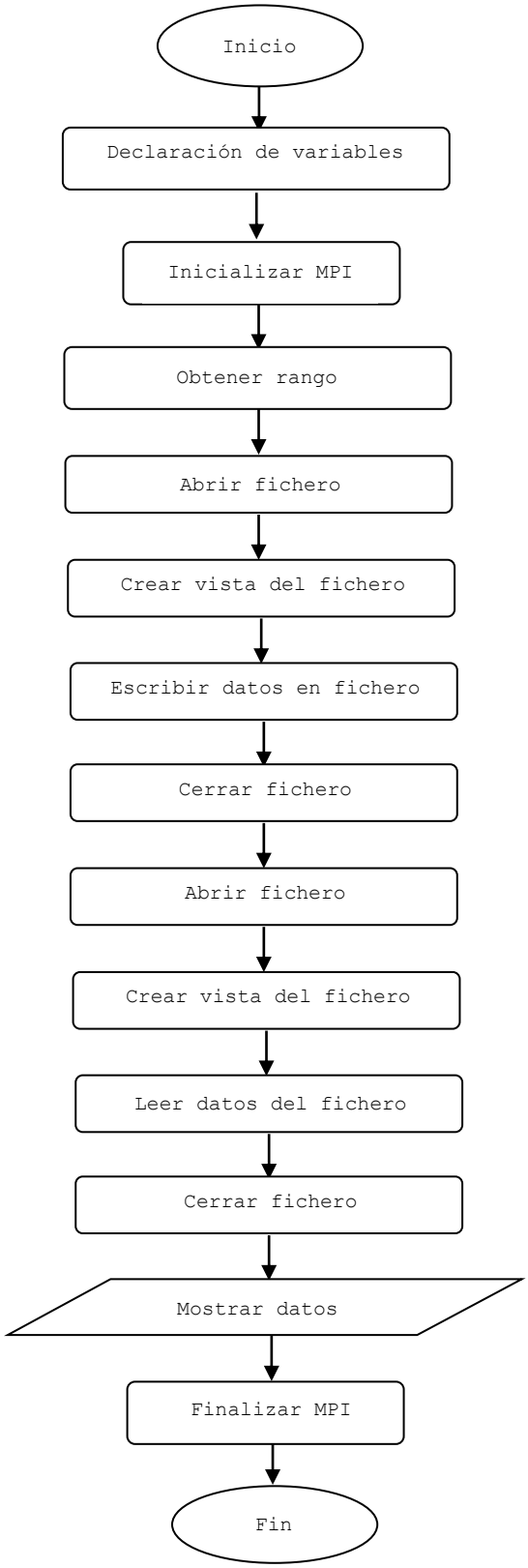
REALIZACIÓN PRÁCTICA

Se va a desarrollar un sencillo ejemplo de escritura y posterior lectura en fichero empleando las funciones de entrada/salida en paralelo. Consistirá en que los procesos lanzados escriban en el fichero su rango un cierto número de veces (definible), uno a continuación de otro y por orden de rango. Posteriormente cada proceso leerá del fichero los datos que introdujo y los mostrará. Para que el fichero pueda ser editable, se puede sumar el valor 48 al número de rango antes de almacenarlo para poder visualizarlo como ASCII. También se puede guardar el rango como carácter.

CUESTIONES

- ¿Se puede pensar en la entrada salida paralela como forma de que un proceso reparta datos a otros alternativamente a las funciones de reparto conocidas?
- ¿Qué inconvenientes plantea esto?
- ¿Puede aportar alguna ventaja?

DIAGRAMA DE FLUJO



RESULTADOS

```
El numero de procesos lanzados es Size=4

El fichero es: fichero.dat
Soy el proceso  0 y mis datos son: 48
Soy el proceso  0 y mis datos son: 48
Soy el proceso  0 y mis datos son: 48
Soy el proceso  0 y mis datos son: 48
Soy el proceso  0 y mis datos son: 48

El tiempo empleado ha sido de 0.016840 segundos

Creando fichero: archivo.txt...
Soy el proceso  2 y mis datos son: 50
Soy el proceso  2 y mis datos son: 50
Soy el proceso  2 y mis datos son: 50
Soy el proceso  2 y mis datos son: 50
Soy el proceso  2 y mis datos son: 50
Soy el proceso  1 y mis datos son: 49
Soy el proceso  1 y mis datos son: 49
Soy el proceso  1 y mis datos son: 49
Soy el proceso  1 y mis datos son: 49
Soy el proceso  1 y mis datos son: 49
Soy el proceso  3 y mis datos son: 51
Soy el proceso  3 y mis datos son: 51
Soy el proceso  3 y mis datos son: 51
Soy el proceso  3 y mis datos son: 51
Soy el proceso  3 y mis datos son: 51
```

PRÁCTICA 6: NUEVOS MODOS DE ENVÍO

OBJETIVOS

- Ampliar el conocimiento de los modos de envío disponibles en MPI.

CONCEPTOS TEÓRICOS

Otros modos de envío en MPI

En la práctica 1 se comentaron a nivel teórico varias formas de establecer la comunicación entre procesos. Se vio de forma práctica el modo de envío estándar que se emplea en MPI mediante las funciones **`MPI_Send`** y **`MPI_Recive`**. En este momento estamos en condiciones de ir un poco más allá y ver qué alternativas se nos plantean y qué sentido puede tener utilizarlas.

Las funciones anteriores trabajan en modo bloqueante, es decir, no permiten que el proceso que las invoca continúe hasta que no se haya completado la operación. Recordemos que en el caso del envío esto no tiene porqué implicar otra cosa que la copia provisional del mensaje enviado en un buffer local.

No obstante este bloqueo aporta potencialmente un retraso en el desarrollo del proceso que tiende a penalizar de manera significativa el rendimiento global de la aplicación. Con el fin de mejorar este comportamiento MPI proporciona una alternativa, incluida también dentro del modo estándar, como es el envío y recepción no bloqueantes. Su funcionamiento se basa en partir la operación, ya sea envío o recepción, en dos partes. En la primera, se inicia la misma y en la segunda se completa. Por el medio se pueden realizar tareas que no tiene sentido retrasar en espera de que se complete la operación. En general, aquellas operaciones que no dependan en absoluto de los datos que se van a intercambiar pueden ir perfectamente en ese espacio intermedio. Al igual que ocurre en las operaciones bloqueantes, la finalización de la operación bloquea el proceso, pero no el inicio.

Veamos en primer lugar las funciones de inicio:

```
int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest, int
   tag, MPI_Comm comm, MPI_Request *request)

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

Como se puede observar, la sintaxis de las funciones es prácticamente idéntica a la que ya conocemos. La diferencia más importante es la presencia del parámetro `*request`. Éste proporciona un puntero a la propia tarea de envío o recepción para poder referenciarla en la operación de finalización.

En lo que se refiere a la finalización de las operaciones, existen dos alternativas: *espera* o *prueba*. Si nos decidimos por la *espera*, se bloqueará el proceso hasta que la operación realmente haya

terminado. Si la alternativa es la *prueba*, el programa sabrá si la operación se ha terminado o no y podrá actuar en consecuencia.

La función de *espera*, tanto para envío como para recepción es:

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Se puede apreciar la utilización que hace del puntero generado en la operación de inicio correspondiente. La función de *prueba* es:

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

A diferencia de la anterior, devuelve un testigo (`*flag`) que si es **1** indica que la operación ha terminado; de lo contrario devuelve un **0**.

REALIZACIÓN PRÁCTICA

Vamos a suponer el siguiente escenario para calcular el factorial de un número: el proceso 0 se encarga de recibir los números que le va introduciendo el usuario. El cálculo se lo encarga al proceso 1 que se supone que reside en una máquina con mayor potencia de cálculo. Para evitar el colapso de las peticiones, deberá hacer esperar al usuario para que no introduzca otro dato hasta que se haya completado el envío anterior. Para ello mostrará un mensaje de espera. Esta situación es poco probable con un hardware de una mínima potencia, pero la incluimos como ejemplo para aprovechar la potencia de las nuevas funciones. También será necesario incorporar una condición de salida. En este caso, el envío del dato 0 indicará que se desea terminar la operación.

CUESTIONES

- ¿De qué manera se puede aprovechar la potencia de las instrucciones vistas en esta práctica para evitar que los procesos trabajen en ocasiones con datos obsoletos?
- ¿Se podría producir una situación de abrazo mortal por estar todos los procesos en curso bloqueados en espera de que se complete una petición?
- Realizar una reflexión sobre el concepto de abrazo mortal indicando cómo afecta a los diferentes modos de envío que se conocen.

DIAGRAMA DE FLUJO

RESULTADOS

```
Proceso 0: Introduce un numero: 4

Proceso 0: >>> Numero (4) enviado!
Proceso 0: recibiendo resultado ...
......
Proceso 0: El resultado es: 4! = 24.000000

Proceso 0: Introduce otro numero: 0

Proceso 0: >>> Numero (0) enviado!
```

PRÁCTICA 7: TIPOS DE DATOS DERIVADOS

OBJETIVOS

- Mejorar la capacidad de intercambio de datos con el empleo de tipos derivados.

CONCEPTOS TEÓRICOS

Tipos de datos derivados en MPI

Hasta el momento, en las funciones de envío y recepción se han intercambiado datos muy simples: enteros, flotantes o vectores. Esto es suficiente en la mayoría de los casos, pero en determinadas ocasiones se hace conveniente disponer de mayor potencia. Imaginemos que queremos pasar de un proceso a otro los datos contenidos en una estructura definida por el usuario. Esa estructura no existe como tipo de datos en MPI, por lo que habría que enviar sus componentes por separado. Esto se puede hacer en todos los casos, pero está claro que no es muy apropiado si existe una opción mejor. La cuestión es, ¿se pueden definir estructuras de datos que entienda MPI? La respuesta es sí, aunque con matices. La posibilidad que proporciona MPI es la creación de tipos de datos derivados. Es un poco más complejo que una estructura de C ya que, no solo se pueden mezclar diferentes tipos de datos de diferentes tamaños, sino que además se puede (se debe) especificar su localización relativa en memoria, por lo que se pueden incorporar datos no contiguos. Esto tiene utilidades muy importantes. Pensemos por ejemplo, que las matrices bidimensionales definidas en C se almacenan en memoria de manera que los datos de una misma fila se encuentran contiguos. De esta manera resulta complicado pasarle a otro proceso una submatriz de una sola vez. Sin acudir a los tipos derivados habría que enviar una vez por cada fila de la submatriz o bien crear una nueva matriz a partir de la original y enviarla completa. Si estos tipos derivados tuvieran que partir de datos contiguos en memoria como ocurre con las estructuras no sería fácil enviar la submatriz, pero al poder especificar desplazamientos relativos el problema queda resuelto.

Veamos de qué herramientas disponemos para crear tipos derivados:

```
int MPI_Type_struct(int count, int *vector_longit_bloques, MPI_Aint
   *vector_desplazam, MPI_Datatype *vector_tipos, MPI_Datatype
   *nuevo_tipo)
```

El parámetro `count` indica el número de bloques del tipo derivado. Los siguientes parámetros son arrays cuya dimensión coincide con `count`, y especifican las características de cada bloque. Así, el parámetro `*vector_longit_bloques` contiene el número de elementos de cada bloque. El parámetro `*vector_desplazam` especifica el desplazamiento en bytes de cada bloque respecto del inicio del mensaje. El parámetro `*vector_tipos` especifica el tipo de datos definido en MPI contenido en cada bloque. Finalmente `*nuevo_tipo` es el puntero al tipo de datos derivado que se acaba de definir.

Después de la llamada a la función anterior, se debe realizar un último trámite para que el nuevo tipo de datos se pueda utilizar. Esto es la llamada a la siguiente función:

```
int MPI_Type_commit(MPI_Datatype *nuevo_tipo)
```

Para acabar de aclarar esta cuestión se dispone a continuación de un sencillo ejemplo de aplicación en el que se crea un tipo de datos derivado, se le asigna valor, se envía y en los procesos receptores se muestra su contenido:

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

int main (int argc, char *argv[])
{
        int mirango;
        int vector_long[3];
        MPI_Aint vector_despl[3];
        MPI_Datatype vector_tipos[3];
        MPI_Datatype nuevo_tipo;

        typedef struct{
                float a1,a2;
                int b1;
                char c1,c2,c3;
        }viejo_tipo;

        viejo_tipo datos;
        vector_long[0] = 2;
        vector_long[1] = 1;
        vector_long[2] = 3;
        vector_despl[0] = 0;
        vector_despl[1] = 2*sizeof(float);
        vector_despl[2] = vector_despl[1] + sizeof(int);
        vector_tipos[0] = MPI_FLOAT;
        vector_tipos[1] = MPI_INT;
        vector_tipos[2] = MPI_CHAR;

        MPI_Init (&argc, &argv);
        MPI_Comm_rank (MPI_COMM_WORLD, &mirango);
        MPI_Type_struct(3,vector_long,vector_despl,vector_tipos,&nuevo_tipo);
        MPI_Type_commit(&nuevo_tipo);

        if(mirango==0)
        {
                datos.a1 = datos.a2 = 1.5;
                datos.b1 = 10;
                datos.c1 = datos.c2 = datos.c3 = 'x';
        }
        MPI_Bcast(&datos, 1, nuevo_tipo, 0, MPI_COMM_WORLD);

        if(mirango!=0)
        {
                printf("a1=%f b1=%d c1=%c\n",datos.a1,datos.b1,datos.c1);
                fflush(stdout);
        }
        MPI_Finalize();
        return 0;
}
```

La función que acabamos de ver permite la máxima flexibilidad a la hora de generar tipos de datos derivados. No obstante, en muchos casos no es necesaria tanta flexibilidad, bien porque los datos

que se van a transferir son todos del mismo tipo, bien porque ocupan posiciones consecutivas. Por este motivo MPI proporciona otras opciones de manejo más simple:

```
int MPI_Type_contiguous(int count, MPI_Datatype viejo_tipo,
   MPI_Datatype *nuevo_tipo)

int MPI_Type_vector(int count, int long_bloque, int salto, MPI_Datatype
   tipo_elemento, MPI_Datatype *nuevo_tipo)

int MPI_Type_indexed(int count,int *vector_longitudes, int
   *vector_desplazam, MPI_Datatype tipo_elemento, MPI_Datatype
   *nuevo_tipo)
```

La primera de las funciones convierte `count` elementos consecutivos del tipo antiguo en el nuevo tipo.

La segunda de ellas convierte `count` bloques con `long_bloque` elementos consecutivos del tipo antiguo, separando cada bloque por una longitud de `salto` elementos.

La tercera toma `count` bloques, donde el iésimo bloque está formado por un número de elementos dado por la iésima componente de `*vector_longitudes` y son del tipo `tipo_elemento`. Este bloque se encuentra desplazado respecto del inicio del nuevo tipo un número de posiciones dado por la iésima componente de `vector_desplazamiento`.

En todos los casos es necesario recurrir a la función **`MPI_Type_commit`** para poder utilizar los nuevos tipos.

REALIZACIÓN PRÁCTICA

Como aplicación de los tipos de datos derivados vamos plantear una situación con tres procesos. El proceso 0 inicializará una matriz de tamaño N×N con números aleatorios y los procesos 1 y 2 inicializarán sendas matrices con valores nulos. Posteriormente el proceso 0 enviará al proceso 1 la matriz triangular superior y al proceso 2 la matriz triangular inferior (figura 7.1). Cada uno de los procesos receptores mostrará su matriz antes y después de recibir los elementos.

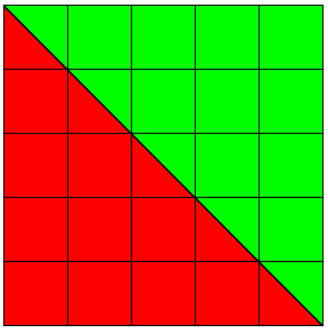


Figura 7.1. Matriz cuadrada dividida en sus respectivas matrices triangulares.

CUESTIONES

- En la práctica realizada ¿qué problemas aparecen si se realiza reserva dinámica de memoria para crear espacio para las matrices? ¿Cómo afecta esto a la definición de nuevos tipos de datos?
- Existen aplicaciones en las que para realizar una operación sobre una submatriz es necesario disponer de los elementos que la rodean (una fila y columna alrededor por ejemplo), aunque no se modifiquen. ¿Cómo podría adaptarse la práctica para que esto sea posible?
- Plantear otras situaciones en las que sea de utilidad la definición de tipos de datos derivados.

DIAGRAMA DE FLUJO

RESULTADOS

```
Proceso 4 de 5: no hago nada!
Proceso 1: Recibo la matriz triangular superior <<<
Proceso 1: Recibidas!
 1  7  4  0  9
 0  8  8  2  4
 0  0  1  7  1
 0  0  0  7  6
 0  0  0  0  2
Proceso 1: Fin
Proceso 0: Iniciando matriz [5]x[5] ...
```

```
 1  7  4  0  9
 4  8  8  2  4
 5  5  1  7  1
 1  5  2  7  6
 1  4  2  3  2
Proceso 0: Envio matriz triangular superior al proceso 1 >>>
Proceso 0: Envio matriz triangular inferior al proceso 2 >>>
Proceso 0: OK!

Proceso 3 de 5: no hago nada!
Proceso 2: Recibo la matriz triangular inferior <<<
Proceso 1: Recibidas!
 1  0  0  0  0
 4  8  0  0  0
 5  5  1  0  0
 1  5  2  7  0
 1  4  2  3  2
Proceso 1: Fin
```

Práctica 8: Gestión Dinámica de Procesos

OBJETIVOS

- Ensayar las técnicas de configuración dinámica del cluster como aproximación al concepto de máquina virtual.

CONCEPTOS TEÓRICOS

Gestión dinámica del cluster

Un potente sistema de gestión dinámica debería permitir arrancar y detener procesos remotos desde un proceso ya iniciado sin restricción alguna. Esto, que está muy bien logrado en PVM, en MPI está aún al principio del camino. Dado que MPI está orientado al concepto de cluster más que al de máquina virtual, no considera como algo preocupante la posibilidad de que algunos de sus nodos se incorporen o desaparezcan en tiempo de funcionamiento. Esto, no solamente limita las posibilidades de aplicación del sistema de paso de mensajes, sino que también puede dar lugar a caídas completas del sistema por desaparición de alguno de sus nodos.

La imparable evolución de MPI está llevando a sus desarrolladores a enfrentarse a estas limitaciones, acercando cada vez más a este sistema de paso de mensajes al concepto de máquina virtual del que partió PVM. El primer paso ha sido añadir algunas funciones que empiezan a tratar con cierto éxito la gestión dinámica de procesos. De esta manera, un proceso en marcha puede lanzar procesos "hijos" que realicen determinadas tareas para él. No es esto una gestión dinámica completa, pero permite ampliar notablemente el campo de aplicación de MPI.

El otro concepto que diferencia la máquina virtual del sistema de paso de mensajes orientado a cluster es la tolerancia a fallos. ¿Qué pasa si un nodo cae? Tradicionalmente MPI no se ha ocupado de este problema, ya que se suponía que en un cluster dedicado al procesamiento en paralelo, todos los nodos trabajaban como uno solo, ubicándose físicamente en el mismo sitio y con un solo usuario. Si esto es así, evidentemente no es probable un fallo en uno de ellos, pero en muchos casos lo que se pretende es aprovechar una red local de propósito general para potenciar algunas aplicaciones. En este tema existe un notable debate entre los desarrolladores de MPI. Hasta el momento, el único avance al respecto lo ha introducido LAM MPI con la posibilidad de arrancar el "demonio" de MPI (*lamd*) en modo de tolerancia a fallos (*lamboot –x*). Esto implica que antes de iniciar una aplicación, el demonio comprobará qué nodos están presentes y según ello asignará procesos. Si un nodo cae en tiempo de funcionamiento, la aplicación completa caerá incluso en este modo de funcionamiento.

Funciones de gestión de procesos en MPI-2

La función principal y casi única es:

```
int MPI_Comm_spawn(char *command, char *argv[], int numprocs, MPI_Info
   info, int root, MPI_Comm comm, MPI_Comm *intercom, int
   array_of_errcodes[])
```

Esta función intentará lanzar múltiples copias del programa MPI apuntado por `*command`, que simplemente es el nombre del programa hijo entrecomillado. El número de copias lanzadas vendrá dado por el parámetro `numprocs`. Mediante `argv` se pueden pasar argumentos al programa lanzado, y si no existen argumentos en línea de comandos para el programa hijo, se puede pasar `MPI_ARGV_NULL`. Mediante el parámetro `info` se proporcionan una serie de valores acerca de dónde y cómo arrancar los procesos hijos, si bien como parámetro `info` se puede pasar `MPI_INFO_NULL`. El parámetro `root` es el rango del proceso padre dentro del comunicador `comm`, devolviendo un nuevo intercomunicador apuntado por `*intercomm` para el intercambio de información entre el padre y los procesos hijo lanzados. Por otro lado, la comunicación entre los procesos hijo es posible, ya que un nuevo `MPI_COMM_WORLD` se crea automáticamente incluyendo exclusivamente a los nuevos procesos. Finalmente, se devuelve un array de posibles códigos de error, uno por proceso lanzado. Para una fácil utilización de la función descrita, como array para los códigos de error se puede pasar `MPI_ERRCODES_IGNORE`.

Una función interesante es:

```
int MPI_Attr_get(MPI_Comm comm, MPI_UNIVERSE_SIZE, int *universe_sizep,
   int *flag)
```

Permite conocer a través del valor devuelto en `*universe_sizep` el número de procesos esperados. Su valor se ajusta en el inicio automáticamente y depende de la implementación de MPI. En LAM viene a coincidir con el número de máquinas configuradas. El contenido de dicho puntero se puede utilizar como parámetro `numprocs` de la función **`MPI_Comm_spawn`**. Si esta funcionalidad no está soportada el contenido de `*flag` será falso (**0**).

Dentro del código de los procesos hijo es necesario utilizar una función que les permita conocer que han sido lanzados por un proceso padre:

```
int MPI_Comm_get_parent(MPI_Comm *intercom)
```

Esta función devuelve a los procesos hijo el intercomunicador `intercom` que les permite intercambiar información con el proceso padre. En caso de que los procesos no provengan de un padre, esta función devuelve el valor `MPI_COMM_NULL`.

No obstante, para poder llevar a cabo la comunicación entre procesos padre e hijo es necesario transformar a ambos lados el intercomunicador en un intracomunicador mediante la función:

```
int MPI_Intercomm_merge(MPI_Comm intercom, int orden, MPI_Comm
   *intracom)
```

Esta función se emplea para crear el nuevo intracomunicador `*intracom` agrupando a los procesos que estaban incluidos en el intercomunicador `intercom`. El parámetro `orden` indica cómo se reordenaran los rangos de los procesos en el nuevo grupo. El grupo de procesos que tiene el valor verdadero (**1**) tiene sus procesos ordenados después del grupo de procesos cuyo valor es falso (**0**). Por ejemplo, si ponemos este parámetro a falso (**0**) en el grupo del padre y a verdadero (**1**) en los hijos, entonces los procesos del grupo del padre comenzarán con el rango 0 en el nuevo intracomunicador, y los hijos quedarán ordenados como estaban en su propio comunicador, pero comenzando por el último rango del grupo anterior. Si en ambos grupos se fija el mismo valor, la elección de qué grupo se ordena primero es arbitraria.

REALIZACIÓN PRÁCTICA

Para la puesta en práctica de los conceptos que acabamos de ver haremos que un proceso "padre" arranque un determinado número de procesos "hijo". Habrá que elaborar dos ficheros ejecutables donde uno de ellos, el "padre" se lanzará desde MPI, mientras que el otro, el "hijo", se lanzará desde el código del "padre". Para comprobar el funcionamiento se mandarán mensajes de saludo: del padre a los hijos y del hijo de menor rango al resto. Cada hijo imprimirá un mensaje de saludo al ser creados, el mensaje de saludo enviado por el padre y el mensaje enviado por el hijo de menor rango.

CUESTIONES

- Según se ha visto que funciona la gestión dinámica de procesos un padre puede lanzar un número determinado de procesos hijos. ¿Sería posible que un hijo tuviera varios padres?
- ¿Podría ser lanzado un hijo por diferentes padres de forma alternativa?
- ¿Puede un hijo lanzar otros hijos suyos?
- Tratar de realizar una reflexión sobre las posibilidades que se abren con estas herramientas.

DIAGRAMA DE FLUJO

RESULTADOS

```
PADRE> Proceso 0 de 1: Cuantos hijos? 3
PADRE> Proceso 0 de 1: Voy a lanzar 3 hijos
PADRE> Proceso 0 de 4: intraCOM
HIJO> Proceso 0 de 3: Soy un hijo!
HIJO> Proceso 1 de 4: intraCOM
HIJO> Proceso 1 de 4 en intraCOM: hola papa!0000
HIJO> Proceso 0 de 3: soy el menor! (hola brother)-(A)-(2006)
HIJO> Proceso 1 de 3: Soy un hijo!
HIJO> Proceso 2 de 4: intraCOM
HIJO> Proceso 2 de 4 en intraCOM: hola papa!0000
HIJO> Proceso 1 de 3: hola brother00
HIJO> Proceso 2 de 3: Soy un hijo!
HIJO> Proceso 3 de 4: intraCOM
HIJO> Proceso 3 de 4 en intraCOM: hola papa!0000
HIJO> Proceso 2 de 3: hola brother00
```

Práctica 9: Ejemplo de Aplicación Práctica

OBJETIVOS

- Demostrar los conocimientos adquiridos desarrollando una aplicación un poco más compleja que permita posteriormente evaluar el rendimiento del sistema.

CONCEPTOS TEÓRICOS

En esta práctica no se van a incluir conceptos nuevos ya que se trata de aprovechar los disponibles. Por supuesto, en este punto no se han visto todas las posibilidades de MPI y la aplicación desarrollada en ningún momento va a ser la mejor de las posibles, pero sí se dispone de conocimientos suficientes para hacer un buen trabajo.

No obstante, puede resultar útil para el desarrollo de la aplicación comentar alguna posibilidad adicional de las funciones que ya conocemos. Concretamente, nos vamos a centrar en la función **`MPI_Recv`**:

```
int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int source, int
   tag, MPI_Comm comm, MPI_Status *status)
```

Esta función proporciona la posibilidad recibir un mensaje sin tener que conocer previamente la fuente ni la etiqueta. Para ello es necesario utilizar el valor `MPI_ANY_SOURCE` como parámetro `source` y el valor `MPI_ANY_TAG` como parámetro `tag`. Posteriormente podemos comprobar el valor de dichos parámetros a través de la estructura del tipo `MPI_Status` que hasta el momento no hemos utilizado. Esta estructura está formada por tres elementos: `MPI_SOURCE`, `MPI_TAG` y `MPI_ERROR`. El primero proporciona la fuente del mensaje que se ha recibido y el segundo proporciona la etiqueta que traía el mensaje; en este caso, si se recibió con el valor `MPI_ANY_TAG`, puede ser interesante conocer la etiqueta que puso el emisor del mensaje ya que se puede codificar algún mensaje en ella. El tercer parámetro proporciona el código de error que se haya podido producir. No vamos a entrar en qué posibles errores se codifican. El interés por el parámetro de estado se encuentra en este momento en la utilidad que puedan proporcionar los dos primeros parámetros y que pueden ayudar a mejorar el código de nuestro programa de aplicación.

REALIZACIÓN PRÁCTICA

Se trata de desarrollar un programa de multiplicación de matrices en paralelo. La forma de realizar el reparto de trabajo entre los procesos es libre. Simplemente se pide que el tamaño de las matrices (cuadradas) pueda ser definido mediante argumento en línea de comando o bien pidiendo su valor por teclado, con el fin de poder realizar pruebas con matrices de diferentes tamaños. Asimismo se debe procurar no limitar el tamaño de las matrices en la medida de lo posible, por lo que se recomienda la reserva dinámica de memoria.

El proceso 0 inicializará las matrices con valores aleatorios de tipo `float` y repartirá la carga de forma adecuada, según el criterio del programador. En una primera fase, se imprimirá el tiempo empleado en la ejecución junto con el resultado en la salida estándar para comprobar que es correcto. Posteriormente se eliminará para permitir que las matrices puedan crecer en tamaño sin hacer esperar por una impresión de datos interminable.

NOTA:

Con el fin de realizar una reserva dinámica de memoria para matrices de un tamaño elevado teniendo la posibilidad de doble indexación es necesario recurrir a los dobles punteros, es decir, definir un array de punteros cada uno de los cuales apunta a una fila de la matriz:

```
// Declaramos un doble puntero para almacenar la matriz
// Esto nos permite referenciar los elementos de la matriz mediante [fila][columna]
float **Matriz;
// Incializamos el doble puntero para almacenar punteros a cada una de las filas de la matriz
Matriz = (float **) malloc(FILAS*sizeof(float *));
// Inicializamos cada puntero de "Matriz" con las posiciones de comienzo de cada fila
for (int i=0; i< FILAS; i++)
{
	Matriz[i] = (float *) malloc(COLUMNAS*sizeof(float));
}
// Ahora ya podemos utilizar la notacion [fila][columna] para nuestra matriz:
for (int i=0; i<FILAS; i++)
{
	for (int j=0; j<COLUMNAS; j++)
	{
		Matriz[i][j] = 0.0;
	}
}
```

Sin embargo, la reserva dinámica de memoria realizada de este modo no garantiza que los datos correspondientes a filas consecutivas sean almacenados de forma consecutiva en memoria, lo cual puede ser imprescindible para algunas funciones de envío. La solución estaría en enviar la matriz fila a fila. Si queremos conseguir un espacio de direccionamiento contiguo manteniendo la posibilidad de la doble indexación para las matrices va a ser necesario inicializar los punteros a las distintas filas de la matriz con valores consecutivos. Esto lo conseguimos del siguiente modo:

```
// Declaramos un doble puntero para almacenar la matriz
// Esto nos permite referenciar los elementos de la matriz mediante [fila][columna]
float **Matriz;
// Incializamos el doble puntero para almacenar punteros a cada una de las filas de la matriz
Matriz = (float **) malloc(FILAS*sizeof(float *));
// Declaramos un puntero para reservar espacio consecutivo para toda la matriz
float *Mf;
// Incializamos el puntero con el tamaño necesario para toda la matriz
// Esto nos garantiza un espacio de memoria en posiciones consecutivas
Mf = (float *) malloc(FILAS*COLUMNAS*sizeof(float));
// Inicializamos cada puntero de "Matriz" con las posiciones de comienzo de cada fila
for (int i=0; i< FILAS; i++)
{
	Matriz[i] = Mf + i* COLUMNAS;
}
// Ahora, ya podemos utilizar la notacion  [fila][columna] para nuestra matriz:
for (int i=0; i<FILAS; i++)
{
	for (int j=0; j<COLUMNAS; j++)
	{
		Matriz[i][j] = 0.0;
	}
}
```

Es muy importante notar que con esta última alternativa, en las funciones de envío se debe utilizar la dirección `Matriz[0]` como dirección de comienzo de los datos almacenados.

CUESTIONES

- Para multiplicar A×B se puede pasar la matriz A completa a todos los procesos y la matriz B se puede repartir por columnas. Pensar otra alternativa.
- ¿Sería posible aprovechar la potencia de la topología cartesiana para facilitar la resolución de este problema?
- El hecho de pasar una de las matrices completa ralentiza notablemente el funcionamiento del programa. Plantear alguna posibilidad en la que no sea necesario trasladar tanta información. Comparar su rendimiento previsible con el de la aplicación que se ha desarrollado.

DIAGRAMA DE FLUJO

RESULTADOS

```
Introduce tamaño de las matrices: 5

La matriz 1 es:
       0.001251      0.563585      0.193304      0.808740      0.585009
       0.479873      0.350291      0.895962      0.822840      0.746605
       0.174108      0.858943      0.710501      0.513535      0.303995
       0.014985      0.091403      0.364452      0.147313      0.165899
       0.988525      0.445692      0.119083      0.004669      0.008911

La matriz 2 es:
       0.377880      0.531663      0.571184      0.601764      0.607166
       0.166234      0.663045      0.450789      0.352123      0.057039
       0.607685      0.783319      0.802606      0.519883      0.301950
       0.875973      0.726676      0.955901      0.925718      0.539354
       0.142338      0.462081      0.235328      0.862239      0.209601

La matriz resultado es:
       1.003332      1.383781      1.320665      1.552783      0.650090
       1.611083      2.132144      2.113359      2.283381      1.182170
       1.133451      1.732278      1.619330      1.514108      0.709936
       0.394984      0.537760      0.522131      0.510089      0.238584
       0.525357      0.921867      0.867681      0.825713      0.665964
Tiempo empleado para el calculo 0.000936 segundos
```

NOTA:
El tamaño de las matrices se puede introducir en tiempo de ejecución (como en el ejemplo anterior) o en línea de comando (recomendado).

Práctica 10: Medida del rendimiento

OBJETIVOS

- Evaluar el rendimiento del sistema de paso de mensajes en diferentes situaciones.
- Adquirir la capacidad de prever la potencia del sistema y extraerla logrando un compromiso entre el esfuerzo de programación y la optimización del código.

CONCEPTOS TEÓRICOS

Rendimiento de un sistema de tamaño variable

A continuación se van a definir una serie de conceptos manejados de forma habitual en la literatura sobre procesamiento en paralelo que permiten evaluar diferentes características de los sistemas y prever su rendimiento. Estos conceptos son los siguientes y se aplican a situaciones donde el número de procesadores utilizados para la ejecución de un programa varía a lo largo del tiempo de ejecución:

- **Grado de paralelismo (DOP)***:* Número de procesadores utilizados para ejecutar un programa en un instante de tiempo determinado. Esta función, **DOP** = *P(t)*, que nos da el grado de paralelismo en cada momento durante el tiempo de ejecución de un programa, se llama **perfil de paralelismo** de ese programa. No tiene porqué corresponder con el número de procesadores disponibles en el sistema (*n*). En las siguientes definiciones supondremos que se dispone de más procesadores que los necesarios para alcanzar el grado de paralelismo máximo de las aplicaciones, **máx**{*P(t)*} = *m* < *n*.

- **Trabajo total realizado**: Si Δ es el rendimiento, velocidad o potencia de cómputo de un solo procesador, medida en MIPS o MFLOPS, y suponemos que todos los procesadores son iguales, es posible evaluar el trabajo total realizado entre los instantes de tiempo t_A y t_B a partir de área bajo el perfil de paralelismo como:

$$W = \Delta \cdot \int_{t_A}^{t_B} P(t) \cdot dt \,.$$

  Normalmente el perfil de paralelismo es una gráfica definida a tramos (figura 10.1), por lo que se puede expresar el trabajo total realizado como:

$$W = \Delta \cdot \sum_{i=1}^{m} i \cdot t_i \,.$$

  En esta expresión t_i es el intervalo de tiempo durante el cual el grado de paralelismo es *i*, siendo *m* el máximo grado de paralelismo durante todo el tiempo de ejecución del programa.

  Según esto, se cumple que la suma de los diferentes intervalos de tiempo es igual al tiempo de ejecución del programa:

$$\sum_{i=1}^{m} t_i = t_B - t_A .$$

- **Paralelismo medio**: Es la media aritmética del grado de paralelismo a lo largo del tiempo ejecución. Se expresa como:

$$\overline{P} = \frac{1}{t_B - t_A} \int_{t_A}^{t_B} P(t) \cdot dt = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{i=1}^{m} t_i} .$$

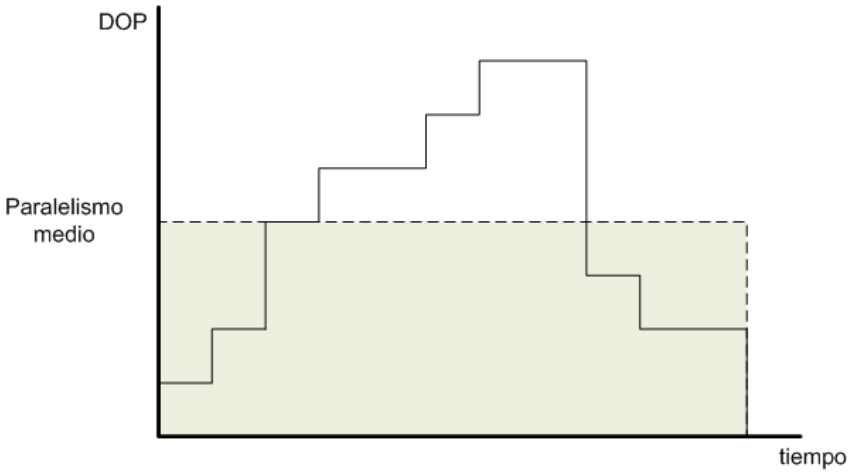


Figura 10.1. Perfil de paralelismo: representación gráfica de P(t).

- **Paralelismo disponible**: Máximo grado de paralelismo extraíble de un programa o aplicación, independientemente de las restricciones del hardware.

- **Máximo incremento de rendimiento alcanzable**: Sea $W_i = i \cdot \Delta \cdot t_i$ el trabajo realizado cuando **DOP** = *i*, de manera que $W = \sum_{i=1}^{m} W_i$.

  En estas condiciones, el tiempo empleado por un solo procesador para realizar un trabajo W_i es $t_i(1) = \frac{W_i}{\Delta}$; para *k* procesadores es $t_i(k) = \frac{W_i}{k \cdot \Delta}$, y para infinitos procesadores es $t_i(\infty) = \frac{W_i}{i \cdot \Delta}$ (sólo trabajan "*i*" procesadores ya que el grado de paralelismo es *i*).

  A partir de aquí se puede definir el **tiempo de respuesta** como:

$$T(1) = \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m} \frac{W_i}{\Delta}$$

$$T(\infty) = \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m} \frac{W_i}{i \cdot \Delta} .$$

El máximo rendimiento que puede proporcionar un sistema paralelo se da cuando en número de procesadores disponibles es ilimitado, por lo que el máximo incremento de rendimiento alcanzable (también denominado *speed-up asintótico*) se obtendrá del cociente entre éste y el rendimiento que proporciona un solo procesador. En términos de tiempos de respuesta se expresa como:

$$S_\infty = \frac{T(1)}{T(\infty)}.$$

Utilizando la definición de $W_i = i \cdot \Delta \cdot t_i$, la expresión del *speed-up asintótico* se puede poner como:

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} \frac{W_i}{\Delta}}{\sum_{i=1}^{m} \frac{W_i}{i \cdot \Delta}} = \frac{\sum_{i=1}^{m} \frac{i \cdot \Delta \cdot t_i}{\Delta}}{\sum_{i=1}^{m} \frac{i \cdot \Delta \cdot t_i}{i \cdot \Delta}} = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{i=1}^{m} t_i} = \overline{P}.$$

Esto se puede simplificar diciendo que el máximo incremento de rendimiento alcanzable por un sistema paralelo que disponga de un número ilimitado de procesadores equivale al paralelismo medio intrínseco de la aplicación que se va a paralelizar. Lógicamente lo complicado es averiguar ese paralelismo intrínseco y lograrlo mediante la programación.

Rendimiento de un sistema con carga de trabajo fija

Existen otras formas de evaluar la ganancia en rendimiento de un sistema. Una de ellas es considerar situaciones donde una carga de trabajo fija se puede repartir entre un diferente número de procesadores. Por ejemplo, decimos que un trabajo *W*, ya sea un programa o un conjunto de ellos, se va a ejecutar en "modo *i*" si para ello se van a emplear *i* procesadores, siendo R_i el rendimiento o velocidad colectiva de todos ellos medida en MIPS o MFLOPS y $T(i) = W/R_i$ el tiempo de ejecución. Así, R_1 sería la velocidad de uno solo y $T(1) = W/R_1$ su correspondiente tiempo de ejecución. Supongamos que el trabajo *W* se realiza en *n* modos diferentes con una carga de trabajo diferente para cada modo, lo cual se refleja en un peso relativo f_i que se le asigna a cada uno:

$$W = W_1 + W_2 + \cdots + W_n = f_1 \cdot W + f_2 \cdot W + \cdots + f_n \cdot W$$

En estas condiciones podemos definir la ganancia en rendimiento como el cociente entre el tiempo $T(1)$ empleado por un único procesador (modo 1) y el tiempo total *T* empleado por los *n* modos (modo 1, modo2,…, modo *n*):

$$S = \frac{T(1)}{T}.$$

El tiempo total de ejecución se calcula a partir del rendimiento y de la carga de trabajo de cada uno de los modos de ejecución:

$$T = \sum_{i=1}^{n} \frac{f_i \cdot W}{R_i} = W \cdot \sum_{i=1}^{n} \frac{f_i}{R_i}.$$

De esta manera, la ganancia en rendimiento se puede poner como:

$$S = \frac{T(1)}{T} = \frac{1/R_1}{\sum_{i=1}^{n} f_i / R_i}.$$

En una situación ideal en la que no existen retardos por comunicaciones o escasez de recursos, considerando un trabajo unidad (W = 1) tendremos que si R_1 = 1, R_i = i :

$$S = \frac{1}{\sum_{i=1}^{n} f_i / i}.$$

En este contexto se enuncia la ley de Amdahl, donde suponemos que el trabajo se realiza en dos modos con pesos relativos α y $1-\alpha$. Una parte se realiza utilizando un procesador tal que $f_1 = \alpha$ y otra en "modo n" con n procesadores tal que $f_n = 1-\alpha$, lo que quiere decir que una parte del trabajo se va a realizar en modo secuencial y el resto empleando la potencia máxima del sistema. En estas condiciones:

$$S = \frac{1}{\frac{\alpha}{1} + \frac{1-\alpha}{n}} = \frac{n}{1+(n-1)\cdot\alpha}.$$

Esto implica que si α = 0, es decir, si idealmente todo el trabajo se puede realizar en "modo n" utilizando la potencia máxima del sistema, entonces:

$$S = n.$$

Sin embargo, si esto no es posible, el máximo incremento de rendimiento alcanzable (*speed-up asintótico*) cuando $n \to \infty$ es:

$$S_\infty = \lim_{n\to\infty} S = \frac{1}{\alpha}.$$

Dicho de otro modo, el rendimiento del sistema se encuentra limitado por el peso de la parte secuencial del trabajo.

Existen algunos parámetros que son de utilidad para evaluar las diferentes características de un sistema paralelo con n procesadores y son los siguientes:

- **Ganancia (*Speed-up*)**: Determina el grado de mejora del sistema:

$$S = \frac{T(1)}{T(n)} \le n.$$

  Se usa para indicar el grado de ganancia de velocidad de una computación paralela.

- **Eficiencia**: Determina el grado de aprovechamiento del sistema:

$$E = \frac{S}{n} = \frac{T(1)}{n \cdot T(n)} \leq 1.$$

La eficiencia mide la porción útil del trabajo total realizado por *n* procesadores. A partir de la eficiencia se puede definir la **escalabilidad**. Así, se dice que un sistema es escalable si la eficiencia *E*(*n*) del sistema se mantiene constante y cercana a la unidad.

- **Redundancia**: Es la relación entre el número total de operaciones que realiza el sistema completo con *n* procesadores y las que realizaría un solo procesador para realizar el mismo trabajo:

$$R = \frac{O(n)}{O(1)} \geq 1.$$

La redundancia mide el grado del incremento de la carga.

- **Utilización**: Refleja el grado de utilización del sistema completo:

$$U = R \cdot E \leq 1.$$

La utilización indica el porcentaje de recursos (procesadores, memoria, recursos, etc.) que se utilizan durante la ejecución de un programa paralelo

- **Calidad del sistema**:

$$Q = \frac{S \cdot E}{R}.$$

La calidad combina el efecto del speed-up, eficiencia y redundancia en una única expresión para indicar el mérito relativo de un cálculo paralelo sobre un sistema.

REALIZACIÓN PRÁCTICA

Se va a evaluar el rendimiento del sistema a partir de la aplicación desarrollada en la práctica anterior. La multiplicación de matrices es un problema de orden cúbico que obliga a una elevada carga de cálculo en cuanto se incrementa un poco el tamaño de las matrices. La medida del rendimiento la vamos a realizar evolucionando en dos sentidos:

- Variando el volumen de cálculo.
- Jugando con el tamaño del sistema.

En lo que al volumen de cálculo se refiere, debemos escoger una serie de valores de tamaño para las matrices que nos resulten representativos. El punto de partida será un valor que provoque un tiempo de ejecución similar para una o varias máquinas en paralelo. Esto depende principalmente de la capacidad de las máquinas disponibles.

A partir de este punto de inicio, se irá incrementando el tamaño de las matrices y con cada nuevo valor se medirá el tiempo de cálculo sobre una máquina, dos, tres, etc. Una representación gráfica de los resultados nos debería llevar a la conclusión de que para un volumen de trabajo muy elevado, el

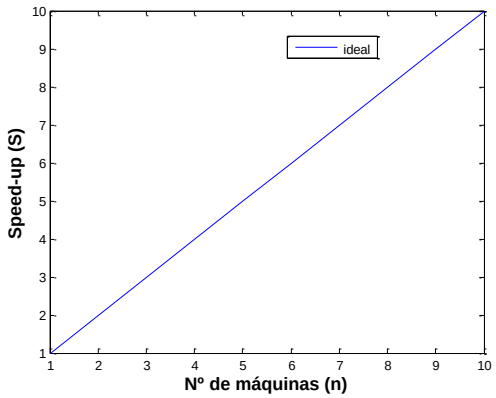
tiempo de proceso debería reducirse proporcionalmente al número de máquinas empleadas si éstas son de similar potencia.

CUESTIONES

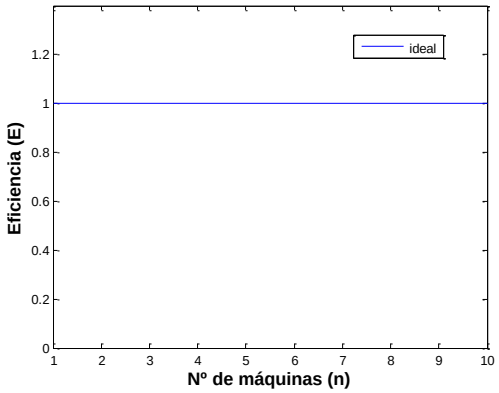
- ¿Está muy lejos el rendimiento obtenido del máximo teórico que se puede alcanzar?
- Evaluar el grado de acercamiento de la aplicación desarrollada a los máximos posibles, calculando para ello de forma aproximada la eficiencia, redundancia, utilización y la calidad del sistema.
- Estimar las causas de la desviación observada.
- Describir qué aspectos se deberían optimizar para obtener un mayor rendimiento.

Gráficas de rendimiento

Representar gráficamente la evolución del speed-up (*S*) y de la eficiencia (*E*) frente al número de máquinas y compararla con la evolución ideal (figura a y figura b, respectivamente).



(a)

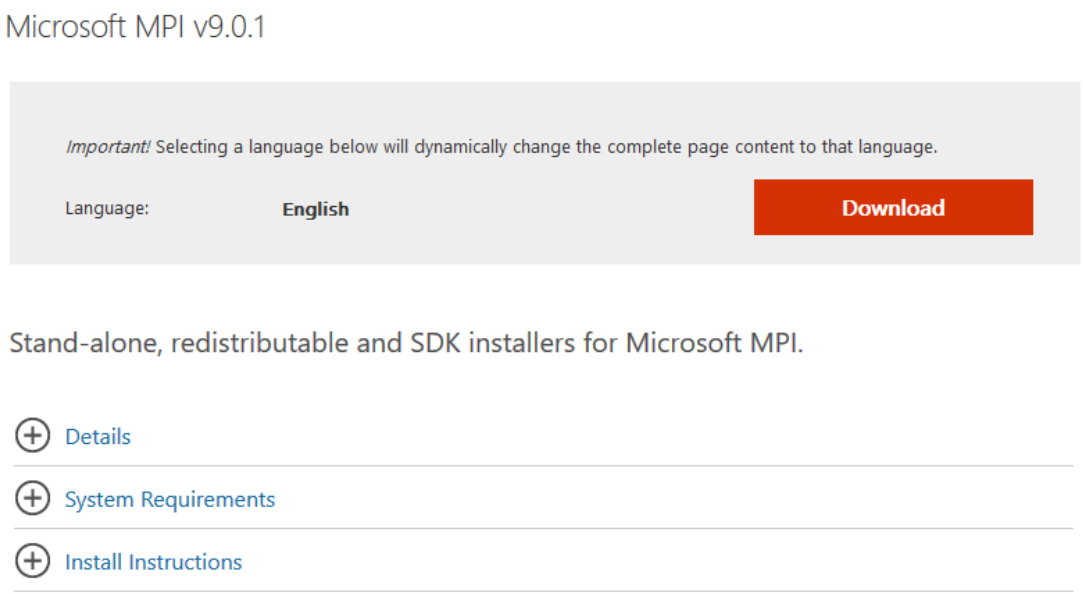


(b)

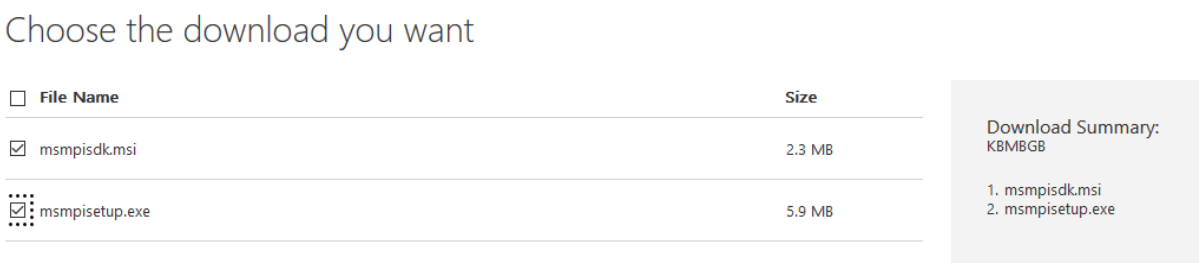
Apéndice: Configuración de MS-MPI.

DeinoMPI es difícil de configurar en algunos sistemas y puede en ocasiones no funcionar. Como alternativa podemos instalar y configurar la distribución de MPI de Microsoft. No nos va a proporcionar una interfaz gráfica, pero podemos hacer lo mismo desde la línea de comandos. Seguiremos los siguientes pasos para ponerlo en funcionamiento:

- Descargar MS-MPI v9.0.1 o la más actualizada, de su localización web:



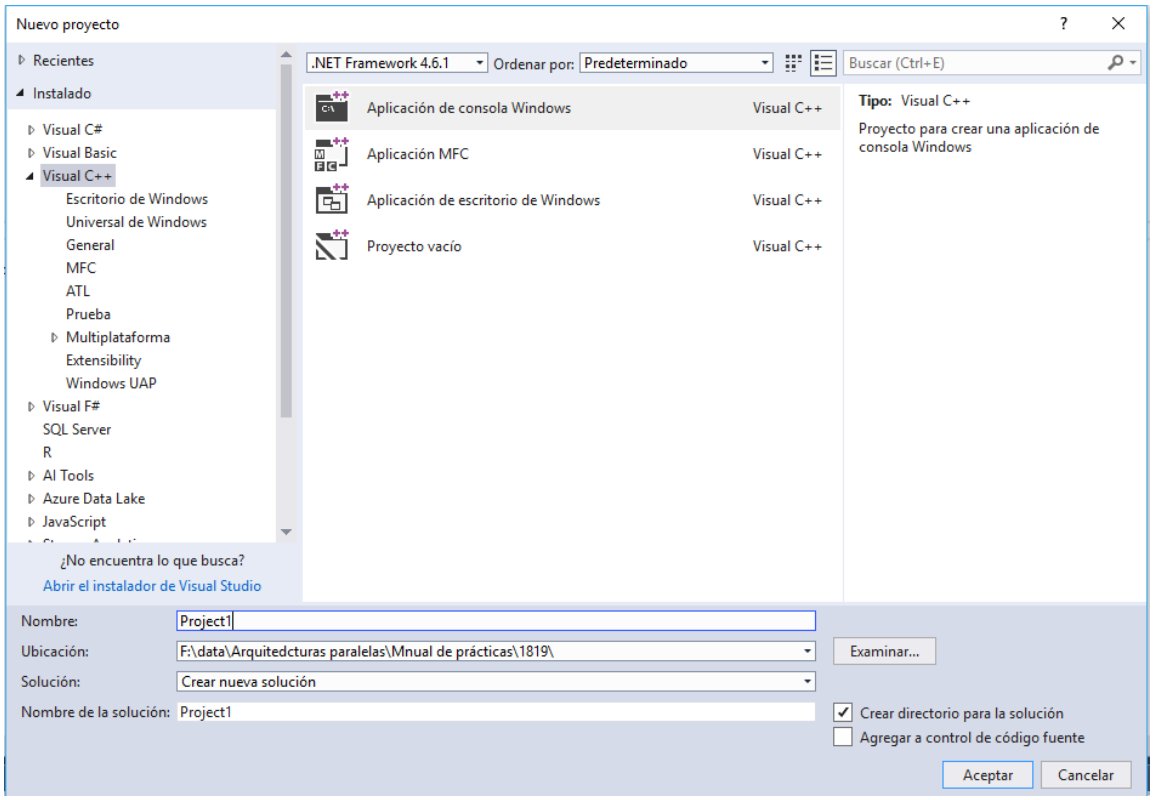
- Hay dos ficheros que descargar e instalar:



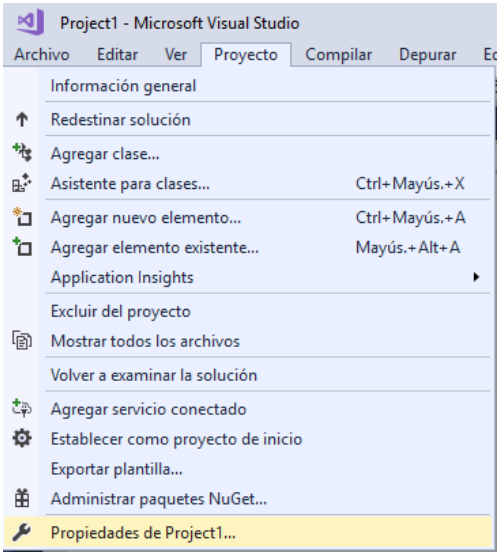
- Cada paquete crea una nueva carpeta: Archivos de Programa > Microsoft MPI y Archivos de Programa (x86) > Microsoft SDKs > MPI.

Ahora ya podemos crear una aplicación. Lo haremos a través de MS Visual Studio siguiendo los pasos que se detallan a continuación:

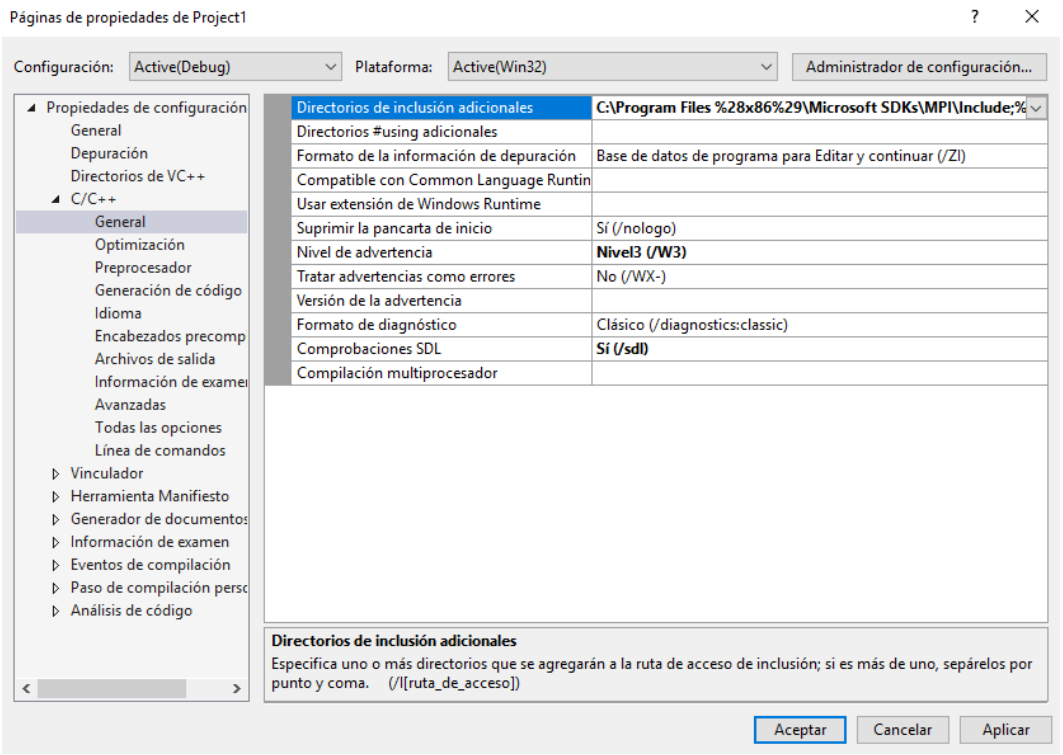
- Crear un **nuevo proyecto y solución**. Se les puede dar a ambos el mismo nombre. Si las opciones que vemos en la figura no aparecen, posiblemetne nos falta algún paquete por instalar. En ese caso acudir al instalador de Visual Sudio 2017 y añadirlo:



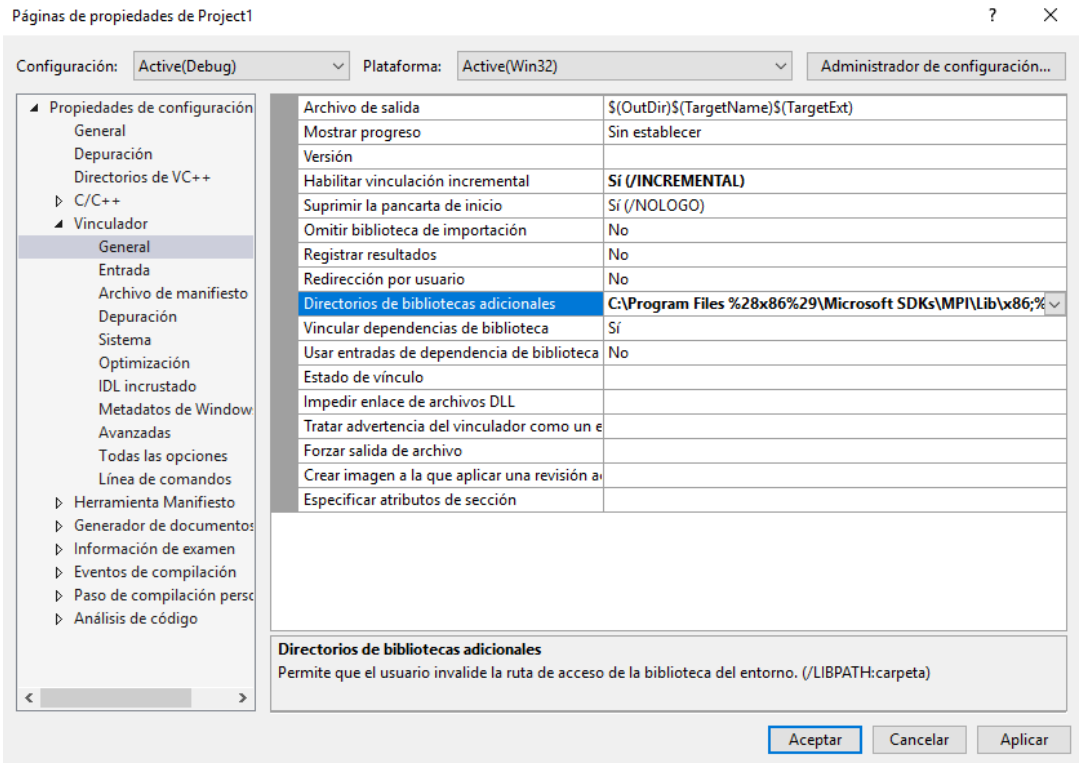
- Ajustar las **propiedades del proyecto** (“*Properties*”):



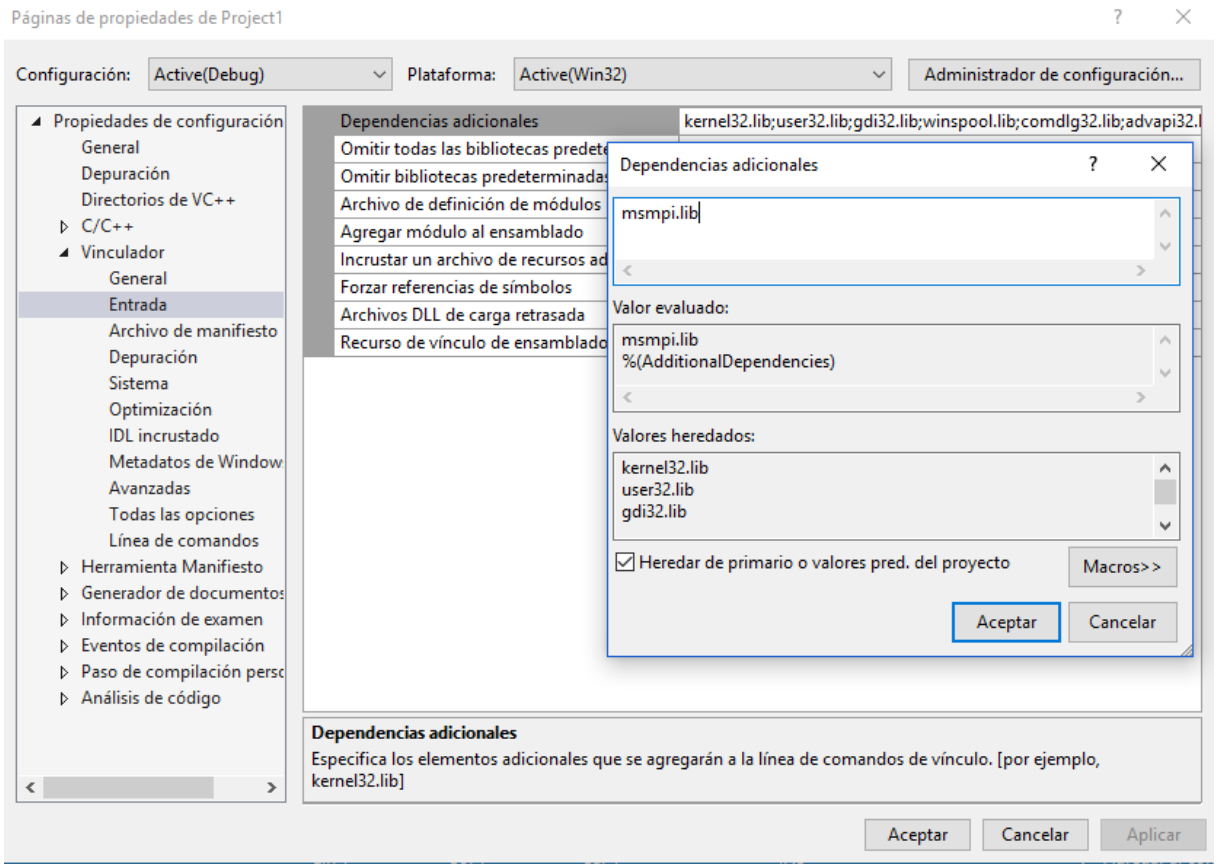
1. Configurar el Nuevo directorio include adicional.



2. De forma análoga configurar la nueva carpeta lib. Es necesario seleccionar la subcarpeta x86 para aplicaciones de 32 bit o x64 para las de 64 bits.



3. Ajustar también la nueva librería MPI.



4. Ahora ya se puede construir la nueva solución como siempre. Para ejecutar el programa, el fichero .exe y el lanzador de MPI deben estar en la misma carpeta o si no se debe ajustar el path para que apunte a la ruta del lanzador. El lanzador es mpiexec.exe y se encuentra en Archivos de Programa > Microsoft MPI > bin. Escribe mpiexec –n np program.exe, donde np es el número de procesos que se pretende lanzar.