Autor

Título

Año

UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR

DE INGENIEROS DE TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

INGENIERO EN ELECTRÓNICA

DESARROLLO DE UN MÓDULO DE APRENDIZAJE PARA PROGRAMACIÓN EN PARALELO

AUTOR: José María Cámara Nebreda

TUTOR: Jesús Manuel Hernández Mangas

18 de Octubre de 2002

TITULO:	**Desarrollo de un módulo de aprendizaje para programación en paralelo**
AUTOR:	José María Cámara Nebreda
TUTOR:	Jesús Manuel Hernández Mangas
DEPARTAMENTO:	Electricidad y Electrónica

Miembros del Tribunal

PRESIDENTE:	Martín Jaraíz Maldonado
VOCAL:	Jesús Manuel Hernández Mangas
SECRETARIO:	Ignacio Martín Bragado
FECHA DE LECTURA:	18 de Octubre de 2002
CALIFICACIÓN:	

RESUMEN DEL PROYECTO

Se ha realizado un estudio de métodos existentes para el aprendizaje de las técnicas de programación en paralelo, aplicables a supercomputadores o clusters de PCs.

Se ha tomado la decisión sobre la solución más apropiada en función de los equipos disponibles y las limitaciones presupuestarias.

Se ha desarrollado un manual de prácticas para el aprendizaje de las técnicas de programación enfocado a la docencia universitaria.

ABSTRACT

Several methods for parallel programming on supercomputers and PC clusters have been tested.

The best solution has been adopted according to equipment availability and economical restrictions.

As a result, a practical course has been developed. It is focused for teaching at University level.

PALABRAS CLAVE

MPI, “programación paralela”, cluster, supercomputador, C, PVM, LINUX.

Índice

Introducción

El presente proyecto final de carrera pretende proporcionar una solución a un problema docente planteado. Se trata de la planificación de un programa de prácticas para la asignatura de Arquitectura e Ingeniería de computadores cuya carga lectiva es de 3 créditos prácticos (30 horas). De acuerdo con el descriptor de contenidos, esta asignatura se dedica a la enseñanza de arquitecturas avanzadas: paralelas y específicas.

A pesar de que los computadores de uso habitual (PCs) incorporan algunas de estas arquitecturas avanzadas, resulta bastante obvio, y así se ha asumido en las escuelas de Ingenieros Informáticos del país, que el grueso del contenido de la asignatura está relacionado con los sistemas paralelos. De esta forma, la parte práctica de la asignatura se dedicaría a la programación de estos sistemas.

El problema que se plantea es que estos sistemas resultan excesivamente costosos como para ser adquiridos como material de prácticas. Será necesario encontrar, por tanto, una solución que permita programar sistemas paralelos a bajo coste.

El proyecto consistirá en llevar a cabo un proceso de estudio de posibilidades, la puesta en marcha de la solución considerada como más apropiada dadas las circunstancias en las que se circunscribe la docencia y la elaboración de una guía para la impartición de las prácticas de programación y para seguimiento de las mismas por parte del alumno.

Desarrollo del proyecto

Como todo proyecto técnico, un proyecto final de carrera se debe planificar correctamente desde el principio para que su desarrollo sea óptimo. A diferencia de la mayoría de proyectos técnicos, en el proyecto final de carrera, la planificación, junto con todo el contenido del proyecto se presenta al final, por lo que inevitablemente mostrará más lo que ha ocurrido realmente que lo que se pretendía que ocurriera.

No obstante, la planificación de un proyecto final de carrera no es tan crítica como en un proyecto real, ya que el incumplimiento de los plazos establecidos no suele llevar asociado un incremento del presupuesto ni da lugar a ningún tipo de responsabilidades en la mayoría de los casos.

En este apartado se proporciona una descripción de las principales tareas que se han desarrollado en la elaboración del proyecto final de carrera y la asignación temporal que ha asumido cada una. Es interesante prestar atención a la descripción de cada tarea para adquirir una idea de la carga de trabajo que supone, ya que la planificación temporal propuesta en el correspondiente diagrama de Gannt responde más a las limitaciones en la dedicación de quien desarrolla el proyecto que a la complejidad de cada labor.

Tarea 1: Estudio Previo

En el momento en que se plantea la necesidad de desarrollar un módulo de aprendizaje para programación en paralelo, se presentan una serie de inconvenientes, algunos de ellos aparentemente insalvables. El ejemplo más evidente es el coste de los computadores de altas prestaciones, habitualmente conocidos como supercomputadores. Estas máquinas justifican su adquisición en base a su aprovechamiento en programas de investigación que exigen una gran potencia de cálculo computacional. De manera accesoria se podrían emplear como medio didáctico, pero difícilmente se pueden adquirir únicamente para ese fin.

La solución a este problema se encontró en la existencia de librerías de paso de mensajes como extensiones a los lenguajes de programación de aplicaciones de cálculo tradicionales (C y fortran) y que se encuentran disponibles tanto para la mayoría de los supercomputadores modernos como para PCs. No sólo eso, sino que la posibilidad de conectar esos PCs formando clusters hace concebir la posibilidad de alcanzar elevadas

potencias de cálculo, de manera que las librerías de paso de mensajes no quedan como una herramienta de simulación de supercomputadores, sino que proporcionan una alternativa viable para aplicaciones reales de investigación.

Este sin duda es el aporte principal del proyecto. Se trata de una solución ideal para un problema inicialmente muy complicado de abordar. Como valor añadido de la solución se puede comentar que las máquinas que van a formar parte de este módulo de aprendizaje no requieren en absoluto una dedicación exclusiva a él, sino que admiten su aprovechamiento para otro tipo de prácticas o incluso para aplicaciones de gestión.

En este momento se puede decir que se ha encontrado una salida para el problema planteado, pero queda por andar todo el camino que se encuentra tras ella. En primer lugar, es necesario documentarse sobre las librerías de paso de mensajes. En este sentido pronto se llega a la conclusión de que existen dos alternativas asumidas universalmente como prácticamente únicas. Esto simplifica enormemente un estudio al que le resta determinar cuál de las dos es mejor. Ya se ha descrito el estudio comparativo realizado, así como sus conclusiones, siendo el desarrollo del mismo el que ha llevado el grueso del tiempo dedicado a esta primera fase del proyecto.

Tarea 2: Adquisición de equipos

Del estudio previo realizado resulta una conclusión acerca de la solución que se pretende adoptar. Se va a basar el módulo de aprendizaje en el desarrollo de prácticas de programación en paralelo empleando el lenguaje C y la librería de paso de mensajes MPI.

Concretamente, se van a desarrollar prácticas en entorno Windows y LINUX. Esto lleva a la conclusión de que va a ser necesario adquirir nuevas máquinas, ya que las disponibles en ese momento, ordenadores Pentium 90 del año 95 no soportan el software que es necesario instalar. En cualquier otro tipo de proyecto, el siguiente paso sería determinar las características de los equipos que se pretende adquirir en función de las necesidades planteadas. En el caso de los ordenadores las cosas funcionan de forma algo diferente. En cada momento existe en el mercado una disponibilidad de componentes que da lugar a una gama ciertamente reducida de modelos para elegir. Ciertamente, existe gran variedad de opciones, pero todas ellas con prestaciones

similares, de forma que se puede jugar un poco con las características del procesador, placa base, disco duro, monitor, etc, pero sin encontrar grandes diferencias. Incluso es posible que lo que se encuentra disponible en el mercado se encuentre muy por encima de las necesidades, pero no habrá opción de reducir el coste a costa de las prestaciones. Se trata por tanto, de comprobar si el material disponible en el mercado de forma ordinaria se adecua a nuestras exigencias. Se comprueba que los requerimientos de capacidad de almacenamiento, memoria y velocidad de CPU que exige el empleo de dos sistemas operativos como Windows 2000 y Linux SUSE 7.3 son abarcados con mucha holgura por la práctica totalidad de las máquinas existentes en el mercado, por lo que se opta por adquirir los modelos que, con cierta previsión de futuro, más se acerquen el presupuesto disponible.

En el anexo de estudio económico, se detallan las características de estas máquinas así como su coste.

El resto de material necesario estaba ya disponible, aunque sería conveniente actualizar el hardware de comunicaciones. Concretamente se debería cambiar el concentrador existente en la actualidad y que no soporta Fast Ethernet por un switch que sí lo soporte. Esta posibilidad se plantea también en el presupuesto.

Tarea 3: Desarrollo del manual de prácticas

Esta es la tarea de más envergadura dentro del proyecto. No plantea dificultades técnicas, de manera que no es previsible ningún retraso significativo por su causa, pero a priori es la que marca el camino crítico en la planificación del proyecto. Por este motivo se ha desarrollado en paralelo con el resto de actividades intentando ponerla en marcha lo antes posible.

Llama la atención su fecha de inicio, anterior incluso a la adquisición de los equipos. No parece lógico en principio que se pueda desarrollar un manual de prácticas sin disponer de equipos en los que probarlas. En este caso esto no es un gran problema ya que las prácticas se pueden programar y probar en un solo equipo, para lo cual cualquier ordenador de sobremesa disponible es adecuado. Lo que funcione en él deberá funcionar en todos (normalmente más rápido).

Gracias a esto, ha sido posible desarrollar la primera parte del manual de prácticas (de la 0 a la 5) en un PC ya disponible. En el momento en que se hizo necesario pasar a programar en LINUX, los equipos nuevos ya estaban operativos.

Tarea 4: Pruebas de configuración e instalación de máquinas

En el anexo sobre instalación de sistemas se exponen todos los detalles interesantes acerca de la configuración de las máquinas que forman parte del cluster.

Esta tarea es la que aporta las dificultades técnicas del proyecto. La instalación de MPI en Windows no supone dificultad, pero en LINUX las cosas son diferentes. La gran cantidad de implementaciones de LINUX existentes y las versiones de cada una de ellas, así como las diferentes implementaciones y sus correspondientes versiones de MPI, hacen que el proceso de configuración de una versión de una implementación en una versión de una determinada implementación del sistema operativo resulte una tarea ciertamente complicada.

A priori, los desarrolladores del software han previsto la gran mayoría de las posibilidades existentes, lo que en principio nos da la esperanza de que se va a conseguir una correcta instalación. Además, los scripts de instalación proporcionados permiten olvidarse del proceso.

El problema viene porque las diferentes posibilidades dan lugar a la necesidad de multitud de parámetros configurables en los scripts, de manera que dar con las opciones que van a posibilitar que el proceso finalice con éxito en un sistema concreto es una tarea muy laboriosa. En muchos casos las opciones por defecto no funcionan y se acaba optando por un método de prueba y error y de búsqueda en Internet de las opciones empleadas por otros usuarios en la misma situación.

En un principio se planteó la posibilidad de generar unos scripts de instalación propios más potentes, pero esto llevaría consigo o bien incrementar el número de parámetros asociados o perder generalidad; todo ello para acortar muy poco el proceso.

La solución adoptada ha consistido en documentar la instalación realizada añadiendo algunos comentarios oportunos para poder trasladar el proceso a otras situaciones.

Tarea 5: Configuración e instalación de máquinas

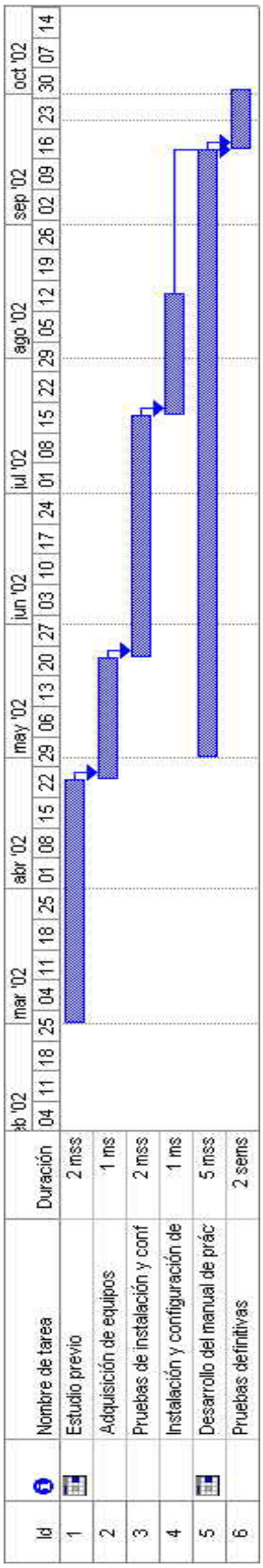
Las dificultades de la tarea anterior conducen, como se ha comentado, a un proceso de prueba y error, o mejor dicho, de instalación y desinstalación. Una vez que se considera que se conoce perfectamente la forma de poner el sistema en marcha, se documenta y se genera el material necesario para reproducirlo de forma rápida. Esto consiste simplemente en la creación de un CD-ROM que contenga todos los archivos instalables, así como las anotaciones que se consideren necesarias para su uso.

Con este material, la presente tarea se convierte en un proceso automático y bastante sencillo, para el cual no se necesita mucho más que tiempo.

Tarea 6: Pruebas definitivas

Una vez que se dispone del sistema completo en funcionamiento y se ha finalizado la elaboración del manual de prácticas, se debe asegurar que todo está dispuesto para comenzar la enseñanza probando las prácticas propuestas en el sistema completo. No planteará problemas importantes, ya que todas las prácticas, auque hubieran sido probadas en una o dos máquinas deben correr en cualquier número de ellas. Algunos pequeños ajustes fueron necesarios en la sintaxis de las prácticas desarrolladas bajo Windows al pasarlas a LINUX debido al empleo de un compilador diferente. Tampoco hubo demasiados problemas en pasar de una implementación a otra en LINUX.

Diagrama de Gannt



Introducción a las técnicas de incremento del rendimiento

Los computadores se han convertido en una herramienta fundamental para la resolución de todo tipo de problemas. Su utilización se remonta a los años posteriores a la Segunda Guerra Mundial, aunque existen precedentes mucho más antiguos.

Desde sus inicios las aplicaciones a las que se ha querido destinarlos han sido cada vez más complejas, de manera que a menudo las prestaciones de las máquinas se han quedado cortas para las necesidades de sus programadores. Esto ha originado desde muy temprano un proceso de búsqueda por parte de los investigadores, de soluciones que permitan incrementar el rendimiento de las máquinas superando las limitaciones impuestas por la tecnología del momento.

Por este motivo, existe un amplio abanico de posibilidades que vamos a tratar de desglosar en este capítulo. Para ello vamos a tratar de proporcionar una clasificación de referencia. Tradicionalmente, las soluciones avanzadas aplicadas en los computadores se han estructurado siguiendo la clasificación de Flynn que establece cuatro categorías de máquinas:

- SISD: procesan un único flujo de instrucciones y un único flujo de datos. Se trata de las máquinas tradicionales a las que se les han ido incorporando algunos avances como la arquitectura solapada y supersolapada (pipe-line y super pipe-line) y/o superescalar.
- SIMD: procesan un único flujo de instrucciones pero un flujo de datos múltiple. Esto quiere decir que trabajan con datos agrupados (vectores). Las máquinas vectoriales comparten esta característica aunque se diferencian de las anteriores en que disponen de una unidad de proceso vectorial en lugar de múltiples unidades escalares.
- MISD: procesan un flujo de instrucciones múltiple sobre un único flujo de datos. Ejecutan por tanto, varias instrucciones de un mismo programa al mismo tiempo sobre los mismos datos. No es una opción muy extendida; tendría reflejo en los denominados arrays sistólicos.
- MIMD: procesan un flujo de instrucciones múltiple sobre múltiples flujos de datos. Se trata de sistemas de procesamiento en paralelo que disponen de varias

unidades de proceso (procesadores) para trabajar cada una de ellas sobre un flujo de instrucciones.

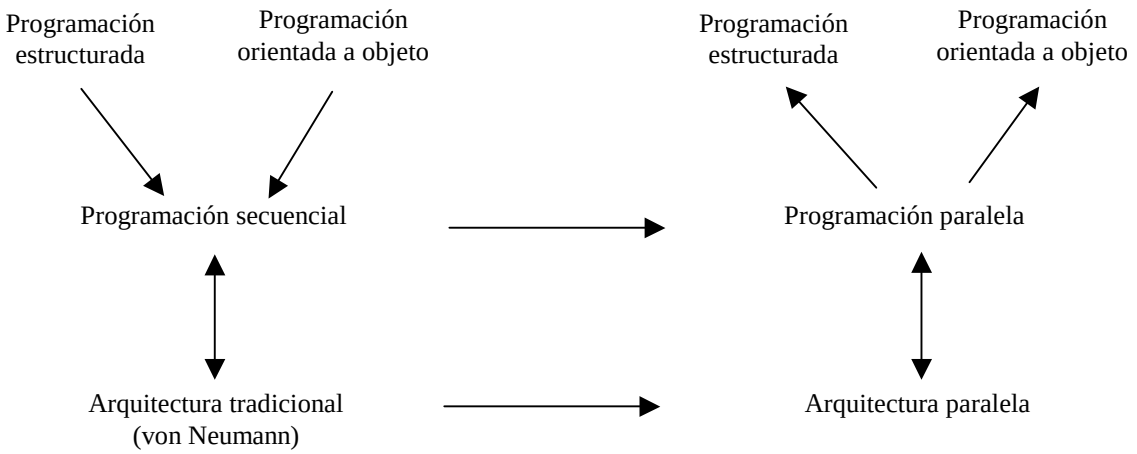
Esta clasificación introduce una nomenclatura que en buena medida se ha convertido en clásica. Sin embargo no resulta una estructura muy afortunada de cara al estudio del paralelismo. El motivo es que se trata de una clasificación que hace referencia a flujos: de instrucciones y de datos. La realidad es que ni las máquinas ni los programas se diseñan atendiendo a estos criterios, por lo que un estudio basado en esta clasificación obliga a mezclar conceptos con excesiva frecuencia. Si lo que se va a abordar es un estudio del hardware de las máquinas sería preferible hablas de modelos de computadores. En este caso nos encontraríamos con dos modelos diferenciados:

- Computadores con arquitectura “von Neumann”, dentro de los cuales se encontrarían desde las máquinas más parcas hasta los procesadores superescalares, supersolapados e incluso vectoriales.
- Computadores paralelos que incluirían principalmente arquitecturas MIMD, aunque también podrían incluirse MISD y SIMD.

Si se desea abordar el estudio de la programación se contemplarían otros dos modelos:

- Programación secuencial en la que existe un único flujo de instrucciones que se procesan una tras otra. Este modelo se emplearía en máquinas SISD, SIMD y vectoriales.
- Programación paralela en la que se pueden procesar diferentes flujos de instrucciones de forma simultánea. Se emplearía en arquitecturas MIMD.

Estos modelos de programación no se corresponden con los paradigmas considerados por la ingeniería del software: la programación estructurada y la programación orientada a objeto. En este caso se trata de dos abstracciones implementables sobre cualquiera de los dos modelos descritos.



El presente proyecto final de carrera se va a centrar en estudiar y poner en marcha algunas de las posibilidades del procesamiento en paralelo, por lo que vamos a entrar a explicar un poco más a fondo las alternativas existentes en el mundo de las máquinas de arquitectura paralela cuya programación es también paralela, es decir, máquinas MIMD.

Máquinas MIMD

Una vez situados en el significado de las siglas MIMD, debemos abordar el estudio de este tipo de máquinas estructurándolas en dos categorías:

- Multiprocesadores: las diferentes unidades de proceso se comunican entre sí a través de una memoria común compartida. Generalmente, cada una tendrá su propia caché local.
- Multicomputadores: cada unidad de proceso dispone de su propia memoria no compartida con el resto. La comunicación entre todas se realizan mediante una red de paso de mensajes.

Antes de seguir adelante es necesario aclarar que cada unidad de proceso dentro de estas arquitecturas puede incorporar otros tipos de soluciones de las vistas en la clasificación de Flynn.

No obstante, aún no se ha llegado a centrar el objeto de este estudio. Dentro de las arquitecturas MIMD tenemos que los sistemas multiprocesador se implementan dentro de una misma máquina, lo que da lugar a los ordenadores multiprocesador. Los sistemas multicomputador pueden estar implementados en una sola máquina o en varias, cada una de las cuales puede tener un procesador o varios. En este último caso, lo que

tenemos es un **“cluster”** de computadores, lo cual sí es finalmente el objeto del proyecto.

El motivo de centrar el estudio en las soluciones “cluster” es la disponibilidad de máquinas de bajo coste y la posibilidad de aprovechar máquinas existentes y que estén dando servicio a otras aplicaciones. Concretamente se trata de emplear ordenadores tipo PC, aunque se pueden añadir otro tipo de maquinas, ya sean de nueva adquisición o disponibles en la actualidad para crear un laboratorio de prácticas de procesamiento en paralelo. Esta solución es aplicable a tareas de investigación en las que se requiera una elevada capacidad de procesamiento. A estos efectos se pueden ir incorporando nuevas máquinas al “cluster”: ordenadores de investigación, de despachos, etc...

De esta forma puede dar la impresión de que vamos a tratar con una pequeña parte de las arquitecturas de computadores que emplean soluciones avanzadas, pero no es así. Ya hemos comentado que cada una de las unidades de proceso pueden incorporar soluciones tales como la estructura supersolapada o una arquitectura interna superescalar e incluso vectorial.

Ahora bien, un sistema multicomputador no es solamente una solución hardware. Para que las aplicaciones se procesen de forma distribuida, hace falta una programación adecuada. Vamos a ver cómo se puede programar una arquitectura de este tipo.

Programación de sistemas multicomputador

Volvamos a remarcar que un sistema multicomputador tiene por objeto incrementar el rendimiento en la ejecución de aplicaciones basándose en la distribución de la carga de procesamiento entre los diferentes nodos de la arquitectura. Es lo que se denomina procesamiento en paralelo. Para que esto sea posible, se debe especificar de alguna manera qué partes de un programa se deben procesar en qué nodos. En nuestro caso cada nodo va a ser una máquina independiente.

Existen dos grandes estrategias en cuanto a la programación de sistemas paralelos:

- Paralelismo explícito: consiste en especificar dentro del código del programa cómo se va a repartir la carga de trabajo.
- Paralelismo implícito: en este caso el programa se no lleva ningún tipo de paralelización, siendo otras capas de software las que se encargan de explicitar el reparto de carga.

Nuestro principal interés se encuentra en el paralelismo explícito ya que aporta un mayor contenido didáctico, además de un incremento de rendimiento potencialmente mayor. No obstante, vamos a hacer referencia a las alternativas existentes, para centrar el estudio posterior en la más adecuada a los objetivos planteados:

- *Sistemas de paso de mensajes.* Constituyen el grueso de nuestro trabajo. Se trata de sistemas de paralelismo explícito cuyo rendimiento depende en gran medida de aspectos bastante aleatorios, teles como la pericia del programador o el tráfico que soporte la red en la que se integran. Esto nos puede llevar a plantearnos su idoneidad a la hora de considerarlos como alternativa; cuestión muy interesante. No obstante, de lo que no cabe duda es de su interés didáctico, que suponen la forma más directa de interacción entre el usuario y el hardware disponible para sus aplicaciones. Tenemos la enorme ventaja de que se trata de sistemas de libre distribución en todos los casos, tanto los desarrollados para entornos Unix como aquellos que corren bajo sistemas operativos de Microsoft.

- *Sistemas dinámicos de reparto de carga transparente.* Constituyen una capa de software que se integra en el sistema operativo y que le faculta para poder realizar un reparto de carga a nivel de proceso entre los diferentes nodos del "cluster". En este caso tenemos un paralelismo implícito, ya que no requieren ningún tipo de consideración a nivel de programación ni tampoco la contemplan. Su rendimiento depende del tipo de aplicaciones que se vayan a ejecutar y nuevamente del tráfico presente en la red. No existe por tanto una dependencia de las habilidades del programador, lo cual, según los casos puede ser una ventaja o un inconveniente. Desde el punto de vista didáctico son soluciones de menor interés, aunque siempre es positivo tratar de realizar una evaluación de sus posibilidades, tanto de forma aislada como trabajando en conjunto con los anteriores. Nuevamente se trata de sistemas de libre distribución trabajando siempre en entornos Unix. Sus principales representantes son MOSIX y CONDOR.

- *Sistemas de explotación de máquinas multiprocesador.* No nos vamos a centrar en las máquinas multiprocesador como tales, pero sí como posibles nodos de un "cluster" junto con otras máquinas mono o multiprocesador. De esta forma, veremos la manera de que el conjunto se beneficie de la potencia que esta categoría de máquinas le puede proporcionar.

Sistemas de paso de mensajes

El paso de mensajes es una técnica empleada para lograr la paralelización del código de un programa adaptándolo al hardware disponible. Cuando un programador pretende que su programa se ejecute en paralelo en diferentes máquinas, debe pensar en primer lugar cómo va a repartir el trabajo. Las técnicas de reparto de carga son complejas y no es algo en lo que se vaya a profundizar pero, a modo de ejemplo, podemos suponer que se pretende programar un procedimiento para evaluar polinomios de grado “n”. Se puede pensar que un reparto de los cálculos entre varias máquinas que se tengan disponibles en un momento dado, puede incrementar notablemente la velocidad de obtención de resultados. Esto es cierto, ahora bien, habría que plantearse cómo repartir el trabajo. Se podría, por ejemplo, ir encargando a cada máquina que calcule un término del polinomio; una de ellas sumaría los resultados parciales obtenidos para obtener el resultado del polinomio completo. Esto es válido, pero también existe la posibilidad de repartir a las diferentes máquinas polinomios completos. Cada una calcula un polinomio completo y devuelve el resultado a aquella en la que se encuentre el usuario. También es válido este planteamiento; ¿cuál es mejor? Esto se va a dejar a criterio del programador; lo que sí está claro es que en cualquiera de los dos casos, las máquinas necesitan comunicarse: enviar datos, ordenar operaciones, proporcionar resultados, etc. Todo ello se va a conseguir mediante mensajes que se van a intercambiar las máquinas a través de la red que les une.

Los mensajes se van a enviar por iniciativa expresa del programa de usuario. Para que ello sea posible será necesario que el lenguaje de programación empleado soporte el intercambio de mensajes. En los sistemas que nos ocupan en este proyecto, los lenguajes de programación inicialmente no lo soportan. Esto es así porque se van a emplear lenguajes de programación no paralelos, lo cual no es ni mucho menos un inconveniente. Por el contrario, la utilización de lenguajes de programación tradicionales permite adaptar programas de aplicación ya programados y desarrollar otros nuevos sin excesivo esfuerzo. Los lenguajes típicamente soportados por estos sistemas son C y Fortran a los que se añaden extensiones (funciones o rutinas) que permiten programar el intercambio de mensajes.

En cualquier caso, la base del sistema de paso de mensajes reside en la existencia de funciones herramientas de envío y recepción, ya sea punto a punto entre dos máquinas o multipunto.

Existen dos grandes alternativas disponibles para la implementación de clusters de paso de mensajes: PVM (parallel virtual machine) y MPI (message passing interface). Se trata en ambos casos de herramientas informáticas que proporcionan las extensiones necesarias a los lenguajes de programación y aplicaciones de gestión de la comunicación entre las máquinas. Uno se puede encontrar con diversas implementaciones, especialmente de MPI, aunque en la mayoría de los casos los organismos que las desarrollan las ofrecen como de libre distribución. Su origen se encuentra en los entornos Unix en los cuales se encuentran más desarrolladas y probadas, aunque la realidad del mercado ha hecho que hayan ido apareciendo implementaciones para sistemas Microsoft.

Las diferencias entre ambos sistemas de paso de mensajes son notables, aunque siempre partiendo de la base de que se trata de dos alternativas válidas para la solución de un problema. El origen de cada sistema es el que marca sus particularidades como veremos con posterioridad en el apartado de conclusiones.

Máquina Virtual Paralela (PVM)

La filosofía de funcionamiento de una máquina virtual consiste en hacer posible la intervención de una serie de computadores individualmente completos en un trabajo colectivo encaminado a incrementar el rendimiento de una determinada aplicación. En este sentido, debe existir una capa de software que se haga cargo de la gestión de procesos para poder iniciar, terminar y coordinar todos los que formen parte de una misma aplicación corriendo en una o varias máquinas. Asimismo se debe de disponer de un lenguaje de programación que facilite las comunicaciones entre los procesos que corren en cada máquina y que forman parte de la aplicación global.

PVM se creó con esta filosofía. Cuenta con un demonio (pvmd) que realiza las labores de gestión de procesos. Existe un demonio corriendo en cada ordenador de la máquina virtual. Desde uno de los ordenadores se inicia el demonio local y los demonios remotos mediante el servicio de shell remoto de TCP/IP (rsh) y a partir de ese momento, todos los ordenadores forman parte de la máquina virtual. Respecto de este servicio hay que decir que su utilización implica la a parición de un problema de seguridad, ya que se está facilitando el acceso a cada ordenador individual desde otras máquinas. Es importante no ejecutar aplicaciones desde cuentas de administrador, ya que con ese nivel de privilegio, se podrían producir verdaderos estragos en el sistema.

La creación de aplicaciones para la máquina virtual paralela sigue la estrategia de emplear extensiones para lenguajes de programación estándar. En este caso, están disponibles extensiones para C y Fortran.

La ejecución de aplicaciones exige disponer de una copia del fichero ejecutable en cada uno de los ordenadores. Todas las copias serán iguales, aunque cada proceso tendrá su identificación dentro de la máquina y el programa deberá especificar qué operaciones debe realizar cada uno. El demonio se encargará de establecer qué procesos van a para a qué ordenadores; si el número de procesos es igual al de máquinas habrá un proceso en cada ordenador; de lo contrario el demonio se encarga de hacer el reparto.

Vamos a dar un repaso a las extensiones que proporciona PVM a los lenguajes de programación para poder explicitar el paralelismo de las aplicaciones. Se hará alusión a las funciones proporcionadas para C entendiendo que existe una rutina equivalente en Fortran para cada función en C, aunque no para todas.

Funciones de gestión de la máquina virtual:

Función	**Operación**
Pvm_addhosts	Añade nuevos ordenadores a la máquina virtual
Pvm_catchout	Captura la salida desde tareas hijas lanzadas desde una tarea padre
pvm_config	Devuelve información actualizada sobre la configuración de la máquina
Pvm_delete	Elimina datos de la base de datos de pvmd
Pvm_delhosts	Elimina ordenadores de la máquina virtual
Pvm_exit	Avisa al demonio local de que el proceso se dispone a abandonar PVM
Pvm_getopt	Proporciona el valor de las opciones de configuración de PVM
Pvm_halt	Apaga la máquina virtual entera
Pvm_hostsync	Devuelve la hora de un ordenador remoto a efectos de sincronización
Pvm_insert	Introduce datos en la base de datos de pvmd
Pvm_kill	Termina un proceso determinado
Pvm_lookup	Recupera datos de la base de datos de pvmd
Pvm_mstat	Devuelve el estado de un ordenador de la máquina virtual
Pvm_mytid	Devuelve la identificación del proceso que la llama
Pvm_parent	Devuelve la identificación del proceso padre que creo al que la llama
Pvm_perror	Proporciona un mensaje que describe el último error devuelto por una llamada a PVM
Pvm_pstat	Devuelve el estado de un determinado proceso
Pvm_reg_hoster	Registra el proceso como iniciador de esclavos
Pvm_reg_rm	Registra el proceso como gestor de recursos
Pvm_reg_tasker	Registra el proceso como iniciador de tareas
Pvm_setopt	Modifica las opciones de configuración de PVM
Pvm_spawn	Arranca procesos hijos del que la llama
Pvm_start_pvmd	Arranca un demonio remoto

Pvm_tasks	Devuelve información sobre las tareas que corren en la máquina virtual
Pvm_tidtohost	Devuelve el ordenador en que está corriendo un determinado proceso

Funciones de señalización:

Función	**Operación**
Pvm_sendsig	Envía una señal a otro proceso
Pvm_notify	Pide notificación de eventos tales como la caída de un ordenador

Funciones de gestión de buffers:

Función	**Operación**
Pvm_mkbuf	Crea un nuevo buffer de mensajes
Pvm_freebuf	Libera un buffer de mensajes
Pvm_getsbuf	Devuelve el identificador del buffer de envío para el envío actual
Pvm_getrbuf	Devuelve el identificador del buffer de recepción para la operación actual
Pvm_setsbuf	Conmuta el buffer de envío activo
Pvm_setrbuf	Conmuta el buffer de recepción activo
Pvm_initsend	Borra el buffer de envío

Funciones de envío:

Función	**Operación**
Pvm_packf	Empaqueta datos para enviar según un formato de impresión
Pvm_pkbyte	Empaqueta para envío datos tipo “byte”
Pvm_pkcplx	Empaqueta para envío datos tipo “float” complejos
Pvm_pkdcplx	Empaqueta para envío datos tipo “double” complejos
Pvm_pkdouble	Empaqueta para envío datos tipo “double”
Pvm_pkfloat	Empaqueta para envío datos tipo “float”
Pvm_pkint	Empaqueta para envío datos tipo “int”
Pvm_pklong	Empaqueta para envío datos tipo “long”
Pvm_pkshort	Empaqueta para envío datos tipo “short”
Pvm_pkstr	Empaqueta para envío datos tipo “string”
Pvm_send	Envía los datos del buffer activo
Pvm_mcast	Realiza un envío tipo multicast a otros procesos
Pvm_psend	Empaqueta y envía datos en una operación

Funciones de recepción:

Función	**Operación**
Pvm_recv	Recibe un mensaje
Pvm_recvf	Redefine la comparación empleada para aceptar mensajes
Pvm_probe	Comprueba si el mensaje ha llegado
Pvm_nrecv	Recibe un mensaje en modo no bloqueante
Pvm_precv	Recibe un mensaje directamente al buffer
Pvm_trecv	Espera indefinidamente la recepción de un mensaje
Pvm_bufinfo	Devuelve información sobre el buffer del mensaje

Pvm_un packf	Desempaqueta datos según un formato de impresión
Pvm_upkbyte	Desempaqueta datos recibidos de tipo “byte”
Pvm_upkcplx	Desempaqueta datos recibidos de tipo “float” complejos
Pvm_upkdcplx	Desempaqueta datos recibidos de tipo “double” complejos
Pvm_upkdouble	Desempaqueta datos recibidos de tipo “double”
Pvm_upkfloat	Desempaqueta datos recibidos de tipo “float”
Pvm_upkint	Desempaqueta datos recibidos de tipo “int”
Pvm_upklong	Desempaqueta datos recibidos de tipo “long”
Pvm_upkshort	Desempaqueta datos recibidos de tipo “short”
Pvm_upkstr	Desempaqueta datos recibidos de tipo “string”

Operaciones de grupo:

Función	**Operación**
Pvm_joingroup	Incluye al proceso que la llama en un grupo
Pvm_lvgroup	Excluye al proceso que la llama del grupo
Pvm_gather	Un proceso recibe mensajes del resto del grupo
Pvm_gsize	Devuelve el número de procesos integrantes del grupo
Pvm_gettid	Devuelve la identificación de un proceso dentro de un grupo
Pvm_getinst	Devuelve el número de instancia de un proceso dentro de un grupo
Pvm_barrier	Bloquea los procesos de un grupo hasta que todos alcanzan esta llamada
Pvm_bcast	Realiza un envío broadcast a todos los procesos del grupo
Pvm_reduce	Realiza una operación de reducción de los datos proporcionado por otros procesos del grupo
Pvm_scatter	Envía un mensaje desde un proceso al resto del grupo

Interfaz de Paso de Mensajes (MPI)

El sistema MPI supone una alternativa que tiende a imponerse como estándar en la programación de aplicaciones paralelas para cluster. No obstante, no considero que sea interesante entrar en una discusión sobre qué sistema acabará por imponerse o si coexistirán ambos. Tampoco las comparaciones que estamos haciendo entre ellos van encaminadas a determinar cuál es mejor. El objetivo es decidir cuál implantar en este proyecto por ser el más apropiado para unas determinadas necesidades.

En el sentido que acabamos de comentar, MPI presenta notables diferencias respecto a PVM. En primer lugar, se trata de un sistema mucho más probado en entorno Windows, lo cual supone una apreciable ventaja ya que el mercado se decanta actualmente por este tipo de plataforma. Solamente funciona con sistemas operativos que se suelen denominar como seguros: Windows NT, 2000 y XP, aunque este último no se ha probado. PVM afirmaba funcionar en todo tipo de entornos Windows, pero en la

práctica no es fiable, al menos en la versión de libre distribución, de manera que en este aspecto MPI lleva la delantera.

Siguiendo con el trabajo en entorno Windows, existe otra notable diferencia, cual es la forma de arranque del “demonio” que gestiona la conexión entre las máquinas. En PVM se arrancaba de forma remota, desde la máquina en la cual se ejecutaba la aplicación. En MPI el “demonio” se arranca de forma automática al iniciar la máquina. A priori esto es una ventaja para PVM, pero se convierte en relativa si se tiene en cuenta que por un lado requiere de un software adicional ya que la “shell” remota no se encuentra implementada en el sistema operativo y por otro lado, presenta un problema potencial de seguridad. Se podría discutir sobre la idoneidad de un sistema de arranque u otro, pero teniendo en cuenta el resto de aspectos comentados, se puede concluir que si queremos trabajar en entorno Windows, MPI se hace más recomendable.

En entorno Unix, la discusión iría a otros terrenos: incremento de rendimiento, potencia del API, soporte y software desarrollado, seguridad, etc. En este sentido, habría que priorizar algún aspecto para decidirse por el sistema más adecuado. Las consideraciones realizadas para el trabajo en entorno Windows siguen pesando, ya que debemos pensar en la interconexión de diferentes plataformas, aunque esto es algo muy complicado debido principalmente a los diferentes sistemas de archivos empleados. Por otro lado, de todos los aspectos valorables: rendimiento, portabilidad, escalabilidad de sistemas y aplicaciones, potencia del API, seguridad etc, MPI destaca a priori por la potencia del API, sin que en otros aspectos existan notables ventajas a favor de PVM. Hay que pensar en las ventajas que puede obtener un programador de aplicaciones informáticas que se quiera adentrar en la programación en paralelo empleando uno u otro sistema. El destino de estas aplicaciones podría estar en la investigación, en cuyo caso el incremento del rendimiento se puede convertir en el parámetro clave; o puede encontrarse en el mercado, en cuyo caso habrá que tener en cuenta además el coste de desarrollo del software. En este último caso la potencia del API es un factor fundamental.

No obstante todas estas consideraciones, vamos a esperar a documentar todos los aspectos del trabajo con MPI y las pruebas realizadas para tomar una decisión definitiva.

A continuación se va a plantear un resumen de la lista de funciones proporcionadas por MPI como extensión al lenguaje C para implementar el sistema de paso de mensajes:

Funciones de control del sistema:

Función	**Operación**
MPI_Abort	Termina todos los procesos asociados a un comunicador
MPI_Finalize	Finaliza el entorno de ejecución MPI
MPI_Finalized	Indica si se ha llamado a MPI_Finalize
MPI_Get_processor_name	Devuelve el nombre de un procesador
MPI_Get_version	Devuelve la versión de MPI
MPI_Init	Inicia el entorno de ejecución MPI
MPI_INIT_thread	Inicia el entorno de ejecución MPI en un entorno multihilo
MPI_Initialized	Indica si se ha llamado a MPI_init
MPI_Pcontrol	Proporciona control de perfiles

Funciones de envío y recepción punto a punto:

Función	**Operación**
MPI_Address	Proporciona la dirección de una posición en memoria
MPI_Bsend	Envío de datos bloqueante con buffer
MPI_Bsend_init	Construye un manejador para un envío con buffer
MPI_Buffer_attach	Crea un buffer para envío de datos
MPI_Buffer_dettach	Destruye un buffer creado para envío
MPI_Cancel	Cancela comunicaciones pendientes
MPI_Get_count	Informa de la cantidad de datos recibidos
MPI_Get_elements	Devuelve el número de elementos básicos en un tipo de datos
MPI_Ibsend	Comienza un envío no bloqueante con buffer
MPI_Iprobe	Comprueba en modo no bloqueante los mensajes que se pueden recibir
MPI_Irecv	Comienza una recepción no bloqueante
MPI_Irsend	Comienza un envío no bloqueante en modo “ready”
MPI_Isend	Comienza un envío no bloqueante
MPI_Issend	Comienza un envío no bloqueante síncrono
MPI_Pack	Empaqueta datos en posiciones contiguas de memoria
MPI_Pack_size	Devuelve el espacio necesario para empaquetar datos
MPI_Probe	Comprueba en modo bloqueante los mensajes que se pueden recibir
MPI_Recv	Recepción de datos bloqueante
MPI_Recv_init	Construye un manejador para recepción
MPI_Request_free	Libera peticiones de envío o recepción persistentes
MPI_Rsend	Envío de datos bloqueante en modo “ready”
MPI_Rsend_init	Construye un manejador para un envío en modo “ready”
MPI_Send	Envío de datos bloqueante
MPI_Send_init	Construye un manejador para un envío estándar
MPI_Sendrecv	Envía y recibe un mensaje
MPI_Sendrecv_replace	Envía y recibe un mensaje usando un buffer
MPI_Ssend	Envío de datos bloqueante con síncrono
MPI_Ssend_init	Construye un manejador para un envío síncrono
MPI_Start	Inicia una comunicación con un manejador persistente
MPI_Startall	Inicia un conjunto de peticiones
MPI_Test	Comprueba si se ha completado un envío o recepción

MPI_Test_cancelled	Comprueba si una petición fue cancelada
MPI_Testall	Comprueba si se han completado todos los envíos o recepciones especificados
MPI_Testany	Comprueba si se ha completado alguno de los envíos o recepciones especificados
MPI_Testsome	Comprueba si se han completado varios de los envíos o recepciones especificados
MPI_Type_commit	Activa un tipo de datos creado
MPI_Type_contiguous	Crea un tipo de datos con datos consecutivos
MPI_Type_extent	Devuelve la extensión (tamaño) de un tipo de datos
MPI_Type_free	Libera un tipo de datos creado
MPI_Type_hindexed	Crea un tipo de datos indexado y con offset
MPI_Type_hvector	Crea un tipo de datos con bloques de datos separados entre sí y con offset
MPI_Type_indexed	Crea un tipo de datos indexado
MPI_Type_lb	Devuelve el marco inferior de un tipo de datos
MPI_Type_size	Devuelve el espacio ocupado por los datos de un tipo de datos (excluyendo desplazamientos)
MPI_Type_struct	Crea un tipo de datos con formato de estructura
MPI_Type_ub	Devuelve el marco superior de un tipo de datos
MPI_Type_vector	Crea un tipo de datos con bloques de datos separados entre sí
MPI_Unpack	Desempaqueta datos en posiciones contiguas de memoria
MPI_Wait	Espera a completar un envío o recepción
MPI_Waitall	Espera a completar todos los envíos o recepciones especificados
MPI_Waitany	Espera a completar alguno de los envíos o recepciones especificados
MPI_Waitsome	Espera a completar varios de los envíos o recepciones especificados

Funciones de comunicación colectiva:

Función	**Operación**
MPI_Allgather	Recoge datos de todos los procesos y los distribuye a todos
MPI_Allgatherv	Recoge datos de todos los procesos y los distribuye a todos en posiciones específicas
MPI_Allreduce	Recolecta datos procedentes de todos los procesos, realiza con ellos una determinada operación y envía el resultado a todos
MPI_Alltoall	Envía datos de todos a todos los procesos
MPI_Alltoallv	Envía datos de todos a todos los procesos en posiciones específicas
MPI_Barrier	Detiene la ejecución hasta que todos los procesos llegan a ella
MPI_Bcast	Realiza un envío desde un proceso al resto de los mismos datos
MPI_Gather	Recolecta datos desde un proceso procedentes del resto
MPI_Gatherv	Recolecta datos desde un proceso procedentes del resto colocándolos en posiciones específicas
MPI_Op_create	Permite crear nuevas operaciones definidas por el usuario
MPI_Op_free	Libera el manejador de una función creada
MPI_Reduce	Recolecta datos desde un proceso procedentes del resto y

	realiza con ellos una determinada operación
MPI_Reduce_scatter	Recolecta datos procedentes de todos los procesos, realiza con ellos una determinada operación y reparte resultados
MPI_Scan	Recolecta datos procedentes de todos los procesos y realiza con ellos una operación diferente en cada proceso
MPI_Scatter	Realiza un reparto desde un proceso al resto de datos distintos
MPI_Scatterv	Realiza un reparto desde un proceso al resto de datos distintos desde posiciones específicas

Operaciones con grupos y comunicadores:

Función	**Operación**
MPI_Attr_delete	Borra el valor de un atributo
MPI_Attr_get	Devuelve el valor de un atributo
MPI_Attr_put	Almacena el valor de un atributo
MPI_Comm_compare	Compara dos comunicadores respecto a contextos, grupos y orden de los procesos
MPI_Comm_create	Crea un Nuevo comunicador que incluye determinados procesos de otro.
MPI_Comm_dup	Copia un comunicador en otro
MPI_Comm_free	Libera un comunicador
MPI_Comm_group	Proporciona el grupo asociado a un comunicador
MPI_Comm_rank	Devuelve el rango de un proceso dentro de un comunicador
MPI_Comm_remote_group	Devuelve el grupo remoto asociado a un intercomunicador
MPI_Comm_remote_size	Proporciona el tamaño de un grupo remoto asociado con un intercomunicador
MPI_Comm_size	Devuelve el tamaño de un comunicador (procesos incluidos)
MPI_Comm_split	Permite reordenar los procesos de un comunicador en otros nuevos
MPI_Comm_test_inter	Comprueba si un comunicador es inter-comunicador
MPI_Group_compare	Compara dos grupos en cuanto a sus procesos miembros y su orden dentro del grupo
MPI_Group_difference	Crea un nuevo grupo con los procesos no comunes a dos anteriores
MPI_Group_excl	Crea un nuevo grupo a partir de otro anterior especificando los procesos excluidos
MPI_Group_free	Libera un grupo creado
MPI_Group_incl	Crea un nuevo grupo a partir de otro anterior especificando los procesos incluidos
MPI_Group_intersection	Crea un nuevo grupo con los procesos comunes de dos anteriores
MPI_Group_range_excl	Crea un nuevo grupo a partir de otro anterior especificando listas de procesos excluidos
MPI_Group_range_incl	Crea un nuevo grupo a partir de otro anterior especificando listas de procesos incluidos
MPI_Group_rank	Devuelve el rango de un proceso dentro de un grupo

MPI_Group_size	Devuelve el tamaño de un grupo de procesos
MPI_Group_translate_ranks	Traduce los rangos de procesos en un grupo a sus rangos en otro grupo
MPI_Group_union	Crea un nuevo grupo a partir de dos anteriores
MPI_Intercomm_create	Conecta dos grupos mediante un interomunicador
MPI_Intercomm_merge	Crea un intracomunicador a partir de un intercomunicador
MPI_Keyval_create	Crea un Nuevo atributo
MPI_Keyval_free	Libera un atributo creado

Funciones de manejo de topologías:

Función	**Operación**
MPI_Cart_coords	Devuelve las coordenadas de un proceso en una topología cartesiana
MPI_Cart_create	Crea un comunicador con topología cartesiana
MPI_Cart_get	Devuelve información sobre la topología cartesiana asociada a un comunicador
MPI_Cart_map	Mapea procesos en una topología cartesiana
MPI_Cart_rank	Devuelve el rango de un proceso en una topología cartesiana
MPI_Cart_shift	Devuelve los rangos de fuente y destino desplazados
MPI_Cart_sub	Divide un comunicador en subredes cartesianas
MPI_Cartdim_get	Devuelve información sobre la topología cartesiana asociada a un comunicador previa a MPI_Cart_get
MPI_Dims_create	Crea una división de procesos en topología cartesiana
MPI_Graph_create	Crea un comunicador con topología específica
MPI_Graph_get	Devuelve información sobre la topología específica asociada a un comunicador
MPI_Graph_map	Mapea procesos en una topología específica
MPI_Graph_neighbors	Devuelve los vecinos de un proceso asociado a una topología específica
MPI_Graph_neighbors_count	Devuelve el número de vecinos de un proceso asociado a una topología específica
MPI_Graphdims_get	Devuelve información sobre la topología específica asociada a un comunicador previa a MPI_Graph_get
MPI_Topo_test	Determina el tipo de topología asociada a un comunicador

Funciones de gestión del entorno:

Función	**Operación**
MPI_Errhandler_create	Crea un manejador de errores
MPI_Errhandler_free	Libera un manejador de errores
MPI_Errhandler_get	Devuelve el manejador de errores de un comunicador
MPI_Errhandler_set	Asocia un manejador de errores a un comunicador
MPI_Error_class	Convierte un código de error en una clase de error
MPI_Error_string	Devuelve un string para un código de error
MPI_Pcontrol	Permite el control de perfiles
MPI_Wtick	Devuelve la precisión del reloj empleado por Wtime

MPI_Wtime	Devuelve el tiempo transcurrido en un proceso

Funciones misceláneas de MPI-2:

Función	**Operación**
MPI_Alloc_mem	Reserva memoria para un buffer de comunicación
MPI_Comm_c2f	Traduce un comunicador C a fortran
MPI_Comm_create_errhandler	Equivalente a MPI_Errhandler_create de MPI-1
MPI_Comm_f2c	Traduce un comunicador de fortran a C
MPI_Comm_get_errhandler	Equivalente a MPI_Errhandler_get de MPI-1
MPI_Comm_set_errhandler	Equivalente a MPI_Errhandler_set de MPI-1
MPI_File_c2f	Traduce un manejador de fichero de C a fortran
MPI_File_create_errhandler	Crea un manejador de errores asociado a un fichero
MPI_File_f2c	Traduce un manejador de fichero de fortran a C
MPI_File_get_errhandler	Devuelve el manejador de errores de un fichero
MPI_File_set_errhandler	Asocia un manejador de errores a un fichero
MPI_Free_mem	Libera memoria previamente reservada
MPI_Get_address	Equivalente a MPI_Address de MPI-1
MPI_Group_c2f	Traduce un comunicador de C a fortran
MPI_Group_f2c	Traduce un comunicador de fortran a C
MPI_Info_c2f	Traduce un manejador de información de C a fortran
MPI_Info_create	Crea un manejado de información
MPI_Info_delete	Elimina un manejador de información
MPI_Info_dup	Crea un manejador de información igual a otro
MPI_Info_f2c	Traduce un manejador de información de fortran a C
MPI_Info_free	Elimina información
MPI_Info_get	Devuelve el valor asociado a una clave
MPI_Info_get_nkeys	Devuelve el número de claves asociadas a un manejador de información
MPI_Info_get_nthkey	Devuelve la enésima clave asociada al manejador
MPI_Info_get_valuelen	Devuelve la longitud del valor asociado a una clave
MPI_Info_set	Añade un par clave-valor a la información
MPI_Op_c2f	Traduce una operación de reducción de C a fortran
MPI_Pack_external	Lee información del buffer en formato “external32” y asigna un buffer para empaquetarla
MPI_Pack_external_size	Lee información del buffer en formato “external32” y calcula el espacio necesario para empaquetarla
MPI_Request_c2f	Convierte una petición de C a fortran
MPI_Request_f2c	Convierte una petición de fortran a C
MPI_Request_get_status	Obtiene el estado actual de una petición en curso
MPI_Status_c2f	Convierte un estado de C a fortran
MPI_Status_f2c	Convierte un estado de fortran a C
MPI_Type_c2f	Convierte un tipo de datos de c a fortran
MPI_Type_create_darray	Crea un tipo de datos a partir de un array distribuido
MPI_Type_create_hindexed	Crea un tipo de datos formado por bloques de datos de diferentes tamaños con desplazamiento entre sí
MPI_Type_create_hvector	Crea un tipo de datos formado por bloques de datos con desplazamiento entre sí
MPI_Type_create_indexed_block	Crea un tipo de datos formado por bloques de datos

	del mismo tamaño con diferentes desplazamientos entre sí
MPI_Type_create_resized	Permite cambiar el tamaño de un tipo de dato
MPI_Type_create_struct	Crea un nuevo tipo de datos con el formato de una estructura predefinida
MPI_Type_create_subarray	Crea un nuevo tipo de datos tomando parte de un array previamente definido
MPI_Type_f2c	Convierte un tipo de datos de fortran a C
MPI_Type_get_extent	Devuelve la extensión de un tipo de datos
MPI_Type_get_true_extent	Similar a la anterior en MPI-2
MPI_Unpack_external	Desempaqueta información obtenida en formato “external 32”
MPI_Win_c2f	Traslada un gestor de ventana de C a fortran
MPI_Win_create_errhandler	Crea un manejador de error que puede ser asociado a una ventana
MPI_Win_f2c	Traslada un gestor de ventana de fortran a C
MPI_Win_get_errhandler	Devuelve el manejador de error asociado a una ventana
MPI_Win_set_errhandler	Asocia un manejador de error a una ventana

Funciones de gestión de procesos:

Función	**Operación**
MPI_Close_port	Libera un puerto de comunicación
MPI_Comm_accept	Permite aceptar conexiones a través de un puerto
MPI_Comm_connect	Permite a un cliente conectarse a un puerto de un servidor
MPI_Comm_disconect	Desconecta al cliente del puerto
MPI_Comm_get_parent	Proporciona un intercomunicacdor con el proceso padre
MPI_Comm_join	Crea un intercomunicador que incluye dos procesos conectados mediante sockets
MPI_Comm_spawn	Lanza procesos hijos
MPI_Comm_spawn_multiple	Lanza procesos hijos con diferentes ficheros binarios
MPI_Lookup_name	Devuelve el Puerto asociado a un servicio
MPI_Open_port	Abre un Puerto de comunicación
MPI_Publish_name	Permite publicar un par puerto-servicio
MPI_Unpublish_name	Elimina un servicio previamente publicado

Funciones de comunicación unilateral:

Función	**Operación**
MPI_Acumulate	Añade los contenidos del buffer de origen al área especificada
MPI_Get	Transfiere información de memoria al proceso de origen
MPI_Put	Transfiere información del proceso origen a memoria remota
MPI_Win_complete	Completa un tiempo de acceso a memoria remota

MPI_Win_create	Crea una ventana en memoria accesible a procesos remotos
MPI_Win_fence	Sincroniza operaciones de acceso remoto sobre una ventana
MPI_Win_free	Libera una ventana de memoria
MPI_Win_get_group	Devuelve el grupo de procesos que tienen acceso a la ventana
MPI_Win_lock	Inicia un tiempo de acceso a una ventana
MPI_Win_post	Inicia un tiempo de acceso a una ventana sin bloqueo
MPI_Win_start	Inicia un acceso a una ventana desde un proceso
MPI_Win_test	Comprueba si se han completado todas las llamadas a MPI_Win_complete
MPI_Win_unlock	Completa el tiempo de acceso a una ventana
MPI_Win_wait	Espera hasta que se han completado todas las llamadas a MPI_Win_complete

Funciones de comunicación colectiva:

Función	**Operación**
MPI_Alltoallw	Proporciona una mayor flexibilidad a la operación de envío de todos a todos
MPI_Exscan	Realiza una operación de reducción sobre datos distribuidos a través de un grupo de procesos

Funciones de interfaz externa:

Función	**Operación**
MPI_Add_error_class	Añadir un nuevo tipo de error a MPI
MPI_Add_error_code	Asocia un código de error a un tipo de error
MPI_Add_error_string	Asocia un string a un código de error
MPI_Comm_call_errhandler	Llama a un manejador de error de un comunicador
MPI_Comm_create_keyval	Sustituye a MPI_Keyval_create de MPI-1
MPI_Comm_delete_attr	Sustituye a MPI_Attr_delete de MPI-1
MPI_Comm_free_keyval	Sustituye a MPI_Keyval_free de MPI-1
MPI_Comm_get_attr	Sustituye a MPI_Attr_get de MPI-1
MPI_Comm_get_name	Devuelve el nombre de un comunicador
MPI_Comm_set_attr	Sustituye a MPI_Attr_put de MPI-1
MPI_Comm_set_name	Asigna un nombre a un comunicador
MPI_File_call_errhandler	Llama al manejador de error asociado a un fichero
MPI_Grequest_compelte	Indica a MPI la finalización de una petición generalizada
MPI_Grequest_start	Inicia una petición generalizada
MPI_Is_thread_main	Indica si el thread que la llama es el principal
MPI_Query_thread	Devuelve el nivel de thread soportado
MPI_Status_set_cancelled	Hace que el estado de una petición indique que ha sido cancelada
MPI_Status_set_elements	Hace que una consulta al estado de una petición devuelva el número de elementos asociados
MPI_Type_create_keyval	Crea una nueva clave de atributo para un tipo de datos

MPI_Type_delete_attr	Elimina un atributo de un tipo de datos
MPI_Type_dup	Duplica un tipo de datos
MPI_Type_free_keyval	Libera una clave de atributo de un tipo de datos
MPI_Type_get_attr	Devuelve un atributo asociado a un tipo de datos
MPI_Type_get_contents	Devuelve los argumentos empleados en la creación de un tipo de datos
MPI_Type_get_envelope	Determina el constructor usado en la creación de un tipo de datos
MPI_Type_get_name	Devuelve el nombre asociado a un tipo de datos
MPI_Type_set_attr	Asocia un atributo a un tipo de datos
MPI_Type_set_name	Asocia un nombre a un tipo de datos
MPI_Win_call_errhandler	Llama al manejador de error asociado a una ventana
MPI_Win_create_keyval	Crea una nueva clave de atributo para una ventana
MPI_Win_delete_attr	Elimina un atributo de una ventana
MPI_Win_free_keyval	Libera una clave de atributo de una ventana
MPI_Win_get_attr	Devuelve un atributo asociado a una ventana
MPI_Win_get_name	Devuelve el nombre asociado a una ventana
MPI_Win_set_attr	Asocia un atributo a una ventana
MPI_Win_set_name	Asocia un nombre a una ventana

Funciones de entrada-salida paralela:

Función	**Operación**
MPI_File_close	Cierra un fichero compartido
MPI_File_delete	Borra un fichero compartido
MPI_File_get_amode	Devuelve el modo de acceso a un fichero
MPI_File_get_atomicity	Devuelve el modo de atomicidad del fichero
MPI_File_get_byte_offset	Obtiene el desplazamiento absoluto de un dato en el fichero
MPI_File_get_group	Devuelve el grupo de procesos que abrió el fichero
MPI_File_get_info	Crea un objeto de información asociado al fichero
MPI_File_get_position	Devuelve la posición del puntero individual al fichero
MPI_File_get_position_shared	Devuelve la posición del puntero compartido al fichero
MPI_File_get_size	Devuelve el tamaño del fichero
MPI_File_get_type_extent	Devuelve la extensión de un tipo de datos del fichero
MPI_File_get_view	Devuelve la vista actual del fichero
MPI_File_iread	Realiza una lectura no bloqueante del fichero
MPI_File_iread_at	Realiza una lectura no bloqueante del fichero con desplazamiento
MPI_File_iread_shared	Realiza una lectura no bloqueante del fichero con puntero compartido
MPI_File_iwrite	Realiza una escritura no bloqueante del fichero
MPI_File_iwrite_at	Realiza una escritura no bloqueante del fichero con desplazamiento
MPI_File_iwrite_shared	Realiza una escritura no bloqueante del fichero con puntero compartido
MPI_File_open	Abre un fichero compartido
MPI_File_preallocate	Asegura que existe espacio para los datos en el fichero
MPI_File_read	Realiza una lectura del fichero

MPI_File_read_all	Realiza un lectura colectiva en el fichero
MPI_File_read_all_begin	Realiza un lectura colectiva en el fichero sin bloqueo
MPI_File_read_all_end	Concluye una lectura colectiva sin bloqueo
MPI_File_read_at	Realiza una lectura del fichero con desplazamiento
MPI_File_read_at_all	Realiza una lectura colectiva con desplazamiento
MPI_File_read_at_all_begin	Realiza una lectura colectiva con desplazamiento sin bloqueo
MPI_File_read_at_all_end	Concluye una lectura colectiva con desplazamiento sin bloqueo
MPI_File_read_ordered	Realiza una lectura colectiva con puntero compartido
MPI_File_read_ordered_begin	Realiza una lectura colectiva con puntero compartido sin bloqueo
MPI_File_read_ordered_end	Concluye una lectura colectiva con puntero compartido sin bloqueo
MPI_File_read_shared	Realiza una lectura con puntero compartido
MPI_File_seek	Actualiza un puntero a fichero
MPI_File_seek_shared	Actualiza un puntero compartido
MPI_File_set_atomicy	Modifica el modo de atomicidad de un fichero compartido
MPI_File_set_info	Asocia un objeto de información a un fichero
MPI_File_set_size	Extiende o trunca un fichero abierto
MPI_File_set_view	Establece una vista individual de un fichero
MPI_File_sync	Actualiza un fichero abierto en un dispositivo de almacenamiento
MPI_File_write	Realiza una escritura en un fichero
MPI_File_write_all	Realiza una escritura colectiva en un fichero
MPI_File_write_all_begin	Realiza una escritura colectiva en un fichero sin bloqueo
MPI_File_write_all_end	Concluye una escritura colectiva en un fichero sin bloqueo
MPI_File_write_at	Realiza una escritura en un fichero con desplazamiento
MPI_File_write_at_all	Realiza una escritura colectiva en un fichero con desplazamiento
MPI_File_write_at_all_begin	Realiza una escritura colectiva en un fichero con desplazamiento sin bloqueo
MPI_File_write_at_all_end	Concluye una escritura colectiva en un fichero con desplazamiento sin bloqueo
MPI_File_write_ordered	Realiza una escritura colectiva con puntero compartido
MPI_File_write_ordered_begin	Realiza una escritura colectiva con puntero compartido sin bloqueo
MPI_File_write_ordered_end	Concluye una escritura colectiva con puntero compartido sin bloqueo
MPI_File_write_shared	Realiza una escritura con puntero compartido
MPI_Register_datarep	Registra una representación de datos

Funciones de conexión entre lenguajes:

Función	**Operación**
MPI_Type_create_f90_complex	Devuelve un tipo de datos que corresponde con un

	complejo
MPI_Type_create_f90_integer	Devuelve un tipo de datos que corresponde con un entero
MPI_Type_create_f90_real	Devuelve un tipo de datos que corresponde con un real
MPI_Type_match_size	Devuelve un tipo de datos correspondiente al tamaño de una variable local

PVM y MPI en el mundo del los computadores

Además del estudio del juego de instrucciones abordado en el apartado anterior, conviene echar un vistazo a la situación de estos sistemas en el mercado actual de supercomputadores. Para ello será necesario establecer cuáles son las soluciones vigentes en la actualidad y cómo se pueden programar. De esta forma se podrá comprobar la vigencia de los sistemas de paso de mensajes en general y de PVM y MPI por separado.

Comencemos por clasificar las alternativas actuales en dos filosofías:

- Soluciones configuradas a base de clusters formados por grandes cantidades de nodos de mayor o menor potencia cada uno. Generalmente se tratará de ordenadores homogéneos dedicados en exclusiva a aplicaciones paralelas.
- Supercomputadores comerciales de elevado coste y prestaciones.

En el primer caso, tenemos una solución equivalente al sistema que vamos a poner en funcionamiento. Por supuesto se tratará de configuraciones mucho más potentes, pero cuyas prestaciones no se traducen en mucha mayor complejidad de configuración, explotación y mantenimiento. A modo de ejemplo se van a aportar un par de casos:

- DAS (distributed ASCI supercomputer): formado por 200 nodos Pentium Pro localizados en cuatro universidades holandesas. Su programación se basa en el paso de mensajes, paro lo cual emplea una implementación de MPI denominada MPI-Panda, basada en MPICH, disponiendo para ella de compiladores de fortran 77 y C.
- VALHAL (Valhala, el paraíso de los vikingos). La traducción no es una mera curiosidad, ya que pone de relieve su filosofía de funcionamiento. Se propone como una alternativa a Beowulf, el cluster de PC basados en LINUX. Dado que Beowulf es el nombre de un guerrero escandinavo, se ha planteado otro nombre

entre histórico y mitológico para esta alternativa. Se encuentra conformada por un cluster de nodos de estaciones Compaq basadas en procesadores Alpha. Emplea como sistema operativo Compaq Tru64 UNIX y permite la programación en C, C++ y fortran mediante sus propios compiladores. La programación paralela es nuevamente posible gracias al paso de mensajes, empleando para ello directamente la implementación MPICH de MPI.

En el segundo caso, estamos hablando de soluciones más tradicionales y en estos momentos de mayor implantación a nivel de grandes centros de cálculo. Existen numerosas compañías que ofrecen supercomputadores, bien de forma exclusiva, bien como un producto más. Vamos a ver los casos de mayor implantación y la forma de programarlos.

- Avalon A12: multiprocesador basado en DEC Alpha 21164. Emplea memoria distribuida y hasta 1680 procesadores. Proporciona una potencia máxima de 1.3 TFLOPS, una memoria de hasta 1.7 TB. Dispone de compiladores para Fortran 77, Fortran 90, HPF y ANSI C. No dispone de ninguna implementación de librerías de paso de mensajes.

- Cambridge Parallel Processing Camma II Plus: se trata de una máquina formada por un array de unidades de proceso específicas de 4096 nodos. Alcanza una potencia máxima de 2.4 GFLOPS. Su memoria es bastante reducida, 512 MB como máximo. La programación de esta máquina se realiza en FORTRAN-PLUS, una implementación propia basada en Fortran 77 con extensiones de Fortran 90 y extensiones propias. También admite C++.

- Compaq AlphaServer SC: es la máquina más potente de Compaq constituida por un máximo de 512 procesadores Alpha 21264a montados en configuración de cluster. Alcanza los 683 GFLOPS y dispone de hasta 2 TB de memoria. Los compiladores existentes admiten Fortran 77, HPF, C y C++. Cada nodo del cluster está formado por 4 procesadores, por lo que a partir de ahí se debe emplear paso de mensajes en la programación. Esto se puede hacer con HPF o en C mediante **PVM** o **MPI**.

- Fujitsu AP3000: máquina de memoria distribuida de hasta 1024 unidades de proceso. Alcanza los 614 GFLOPS con una memoria máxima de 2 TB. Se puede programar en: Parallel fortran/AP, Fortran 90, HPF, C y C++. Dispone de

implementaciones propias de **PVM** y **MPI**, denominadas PVM/AP y MPI/AP respectivamente.

- Hitachi SR8000: máquina de memoria distribuida basada en procesadores PowerPC con un máximo de 512 nodos. La máxima potencia alcanzable llega a 6.1 TFLOPS y la memoria hasta 8 TB. La programación se puede realizar en Fortran 77, Fortran 90, Parallel fortran, HPF, C y C++. Soporta **PVM** y **MPI**.
- HP Exemplar V2600: máquina de memoria distribuida con hasta 32 nodos PA-RISC 8600. Alcanza con esta configuración una potencia de cálculo de 291 GFLOPS y una capacidad de memoria de 128 GB. Se puede programar en: Fortran 77, Fortran 90, Parallel fortran, HPF, C y C++. No prevé el empleo de librería de paso de mensajes estándar.
- IBM RS/6000 SP: máquina basada en procesadores RS/6000 dispuestos en cluster con un número máximo de 2048 nodos. Alcanza los 3.07 GFLOPS y dispone de una memoria de hasta 1 TB. Se puede programar en Fortran 90, HPF, XL C y C++. Dispone de implementaciones propias de **PVM** y **MPI**.
- NEC Cenju-4: máquina de memoria distribuida basada en procesador MIPS R10000 de los que puede montar hasta 1024. Llega a los 410 GFLOPS con un máximo de 512 GB de memoria. Admite programación en Fortran 77, Fortran 90, HPF y ANSI C. Admite **MPI**.
- Quadrics Apemille: es un array de procesadores de hasta 2048 nodos. Llega a 1TFLOP con una memoria máxima de 64 GB. Es programable en una versión de Fortran bastante propietaria.
- SGI Origin: máquina de memoria distribuida basada en el procesador MIPS R10000, de los cuales puede tener hasta 128. Alcanza 76.8 GFLOPS y dispone de hasta 512 GB de memoria. Es programable en Fortran 77, Fortran 90, C, C++, ADA y Pascal. Dispone de implementaciones propias de **PVM** y **MPI**.
- Sun E10000 Starfire: máquina de memoria distribuida basada en el procesador UltraSPARC con un máximo de 64 nodos. Su potencia máxima de cálculo es de 51.2 GFLOPS y su memoria de 64 GB. Es programable en Fortran 77, Fortran 90, HPF, C y C++. No soporta librerías de paso de mensajes.

- Cray SV1: multiprocesador de memoria compartida. Su arquitectura soporta un máximo de 1024 nodos, los cuales son unidades de proceso vectoriales. Llega a 1.2 TFLOPS y a 1 TB de memoria. La programación se puede realizar en Fortran 90, C, C++, Pascal y ADA. Admite **MPI**.
- Tera MTA: máquina de memoria distribuida de hasta 256 unidades de proceso. Llega a alcanzar los 256 GFLOPS y 256 GB de memoria. La programación se realiza en Fortran, HPF, ANSI C o C++. No soporta librerías de paso de mensajes.

Como conclusión de este repaso realizado en las dos vertientes consideradas se puede extraer la oportunidad de la solución propuesta en el presente proyecto. Las librerías de paso de mensajes consideradas estándar se están imponiendo a nivel de programación tanto de los denominados super-clusters como de supercomputadores que podemos considerar como soluciones tradicionales. Vemos que en el caso de los cluster la solución adoptada es MPI como estándar asumido. En el caso de las super-máquinas, tanto PVM como MPI (más MPI) conviven con opciones tradicionales como Fortran 90 o HPF, además de lenguajes con extensiones propietarias, pero cada vez se aprecia mejor la tendencia a incorporar estas librerías en las máquinas más modernas.

Otra cuestión es la discusión de moda sobre si los super-cluster van a desplazar al resto de máquinas de cálculo. En tal caso la tendencia sería más clara hacia el empleo de las librerías de paso de mensajes como opción única, sobre todo hacia MPI.

Sea como fuere se demuestra que el aprendizaje de las técnicas de programación en paralelo basadas en el empleo de librerías de paso de mensajes y, por tanto, la aportación que se realiza en este proyecto para el aprendizaje de MPI, pone al alumno en una posición de vanguardia en las técnicas de programación en paralelo y le capacita para poder crear aplicaciones para cualquier máquina de cálculo con el mínimo esfuerzo. Es evidente que en un curso de diez prácticas no se va a alcanzar una pericia extraordinaria en la programación de aplicaciones; esto se deberá adquirir principalmente a base de práctica.

Conclusiones

A partir del estudio de las tablas de funciones incluidas en PVM y MPI y de la experiencia de trabajo con ambos sistemas se puede establecer una comparación de

posibilidades entre ambos que sirva de referencia para tomar una decisión sobre cuál de ellos puede ser el más adecuado para implantar y desarrollar aplicaciones.

En una primera aproximación se aprecia que MPI incluye un volumen de funciones muy superior a PVM. Esto por sí mismo no indica nada. Recordemos como ejemplo la eterna disputa entre las máquinas RISC (de juego de instrucciones reducido) y las máquinas CISC (de juego de instrucciones complejo) por cuál de las dos alternativas tecnológicas resulta más potente. En principio, un mayor volumen de funciones implica mayor facilidad para el desarrollo de código; una vez que se conocen las funciones, claro. La diferencia importante estaría en que las funciones"extra" permitieran realizar tareas imposibles para PVM. Esto efectivamente es así, por lo que sí existen diferencias de fondo el las posibilidades de cada sistema. No hay que engañarse pensando que todo van a ser ventajas para MPI ya que también hay cosas que PVM, a pesar de disponer de muchas menos funciones permite hacer otras cosas que MPI no permite. Será mejor desglosar un poco el estudio de posibilidades para facilitar la comprensión de las diferencias.

Es posible clasificar las diferencias en dos apartados: bloques de funciones que proporcionan una funcionalidad no disponible en el otro sistema y diferencias entre funcionalidades presentes en ambos sistemas. En el primer caso, la discusión es mucho más obvia:

- Entrada-salida paralela: las funciones de gestión de ficheros compartidos existentes en MPI no aparecen en PVM: Esta es una funcionalidad no presente en PVM.
- Tipos de datos derivados: aparte de las posibilidades de empaquetado y desempaquetado de datos; el manejo de tipos de datos derivados proporciona una gran potencia a la hora de distribuir información entre procesos. Esto no está disponible en PVM.
- Topologías virtuales: otra posibilidad interesante para facilitar la programación de ciertas aplicaciones. Tampoco está disponible en PVM.
- Tolerancia a fallos: muy importante sobre todo cuando se están corriendo aplicaciones de larga duración cuya caída supone la pérdida de muchas horas de trabajo.

Un poco más complicado resulta discutir sobre funcionalidades disponibles en ambos sistemas aunque con diferencias más o menos apreciables. Vamos a tratar de comentar los aspectos en los que las diferencias son más claras:

- Gestión dinámica: la filosofía de MPI-1 se basaba en el empleo de grupos estáticos, excluyendo la posibilidad de una gestión dinámica de procesos. MPI-2 manteniendo esa filosofía introdujo la posibilidad de lanzar nuevos procesos desde uno que ya está en marcha. A pesar de ello y, y teniendo en cuenta la ausencia de tolerancia a fallos, la gestión dinámica de procesos en MPI es mucho más básica que en PVM.

- Operaciones no bloqueantes: en todos los aspectos relacionados con el intercambio de información entre procesos MPI proporciona un mayor número de funciones y una mayor potencia. No obstante, es en la comunicación no bloqueante donde se aprecia más la diferencia. Esta opción, entendida como posibilidad de que un proceso no se vea bloqueado por una operación de envío o recepción, no está disponible como tal en PVM. Esto no implica que haya aplicaciones no implementables en PVM por este motivo, pero si supone un handicap para conseguir un software fiable en el que la posibilidad de “abrazos mortales” quede prácticamente excluida.

Hasta aquí se ha llegado a una comparación superficial situada en el punto de vista del usuario. Resulta interesante conocer la opinión de expertos e implementadores sobre el particular a fin de obtener una explicación más técnica de las diferencias el incluso llegar a conocer sus causas. A continuación se va a exponer este punto de vista logrado a partir fundamentalmente de la lectura de dos artículos de gran calidad:

PVM and MPI: a Comparison of Features

Goals guiding design: PVM and MPI

De acuerdo con las opiniones vertidas en estos artículos, las diferencias entre los dos sistemas derivan en primera instancia de sus respectivos conceptos de partida. PVM como su nombre indica está orientado al concepto de máquina virtual. El nombre de MPI por el contrario no resulta muy aclarativo, ya que sistema de paso de mensajes son ambos. Se puede decir que MPI está orientado al concepto de cluster o de máquina multiprocesador. La diferencia fundamental entre el concepto al que están orientados cada uno se origina en la disposición de las máquinas. Una máquina virtual de forma

genérica se puede considerar como un sistema formado por un conjunto relativamente grande de máquinas que pueden ser heterogéneas conectadas por un hardware de red y que no tienen porqué estar necesariamente próximas ni dedicadas en exclusiva a la computación en paralelo. Una máquina multiprocesador consiste en una sola máquina que dispone de varios procesadores en su interior; un cluster sería una máquina virtual formada por ordenadores dedicados en exclusiva a la computación en paralelo y ubicados muy próximos.

A partir de estos conceptos podemos entender, por ejemplo, que MPI no se planteara la tolerancia a fallos como uno de sus objetivos de partida. Inicialmente se planteó como objetivo el obtener el máximo rendimiento del sistema disponible así como una calidad en el software generado que permitiera la portabilidad de las aplicaciones de un entorno a otro sin que se debieran temer fallos por bloqueos, abrazos mortales, etc. Esto, como veremos a continuación le alejó aún más de aspectos como la gestión dinámica de procesos o la tolerancia a fallos.

La consecución de la seguridad de las comunicaciones hace necesaria una modularidad de las aplicaciones que impida que, mensajes enviados de forma explícita por los programas de usuario o implícitamente por funciones por ellos invocadas, sean recibidos por otras funciones o programas a los que no iban dirigidos, derivando en un mal funcionamiento de las aplicaciones. A pesar de que MPI proporciona la posibilidad de emplear etiquetas en los mensajes, esto no soluciona completamente el problema ya que las librerías de funciones pueden generar todavía un conflicto con los programas de usuario.

Para eliminar completamente el problema, MPI implementa contextos en los que inscribe a todos los actores que intervengan en la comunicación. De esta forma sólo es posible el intercambio de información dentro de un determinado contexto e imposible un conflicto con otros contextos. De cara a la programación, el usuario dispone de un comunicador que identifica su correspondiente contexto. Este sistema exige una composición estática de los contextos, lo que entra en conflicto, como se dijo anteriormente con la gestión dinámica y la tolerancia a fallos. En el momento en que un proceso desaparece el contexto cae y sus consecuencias sobre la aplicación son indefinidas y en todo caso dependientes de la implementación de MPI en la que se esté trabajando.

MPI-2 incorpora una cierta funcionalidad de configuración dinámica, permitiendo el lanzamiento de procesos hijos desde un proceso padre. Para ello permite la generación de nuevos contextos, aunque no especifica nada más que pueda dar pie a implementar un cierto grado de tolerancia a fallos, por lo que la caída de un proceso, padre o hijo degenera en un fallo irrecuperable de la aplicación completa.

Otros conceptos interesantes que podemos examinar de forma breve son los de portabilidad, heterogeneidad e interoperabilidad; conceptos todos ellos considerados como objetivos a lograr en ambos sistemas:

- La portabilidad implica que un software desarrollado en un determinado entorno de trabajo pueda funcionar correctamente en otro. En principio PVM y MPI son portables porque sus especificaciones no tienen ninguna dependencia de las máquinas.
- La heterogeneidad implica la presencia de máquinas distintas en el sistema. Esto puede plantear algún problema por los distintos formatos de datos que pueden emplear. Quien mejor aborda este problema es MPI por la presencia de tipos de datos propios que son independientes de la arquitectura.
- La interoperabilidad es uno de los problemas de MPI. En principio, aunque la mayoría de las aplicaciones desarrolladas bajo una implementación van a funcionar en otra, otra cosa es la integración en un mismo entorno de máquinas en las que corran diferentes implementaciones. Va a ser muy complicado que se entiendan entre sí. Por la experiencia adquirida en este trabajo, MPICH y LAM-MPI no son capaces de convivir juntas, aunque programas desarrollados en una sí funcionan en otra, salvo que empleen gestión dinámica de procesos no disponible en MPICH en la actualidad.

Las conclusiones que se obtienen después de un estudio comparativo de ambos entornos de programación en paralelo no permiten lanzar una afirmación rotunda sobre cuál es mejor, pero sí permiten tomar una decisión razonada.

MPI es un entorno más actual y dinámico; su evolución en este momento es imparable, existiendo multitud de organismos, principalmente universitarios trabajando en distintas implementaciones. Dispone de una librería de funciones mucho más potente que PVM, pero se queda corto en la gestión dinámica de procesos y no implementa tolerancia a fallos.

Por los motivos anteriores, se considera que MPI es más apropiado para lograr un software de mayor calidad, pero en aplicaciones que vayan a trabajar sobre un hardware que responda al concepto de máquina virtual, y más cuanto mayor sea el número de nodos y la distancia entre ellos, PVM por ser tolerante a fallos resulta más recomendable.

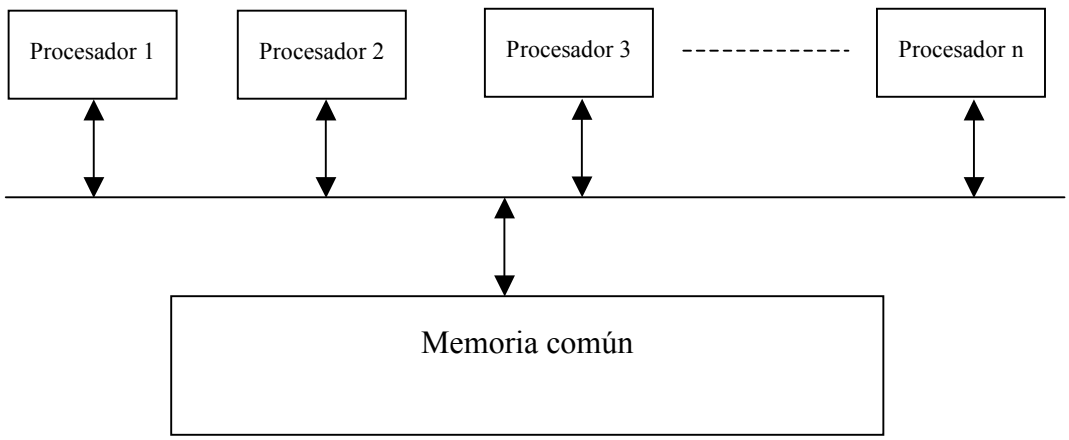
La visión proporcionada sobre la implantación de MPI y PVM a nivel de grandes máquinas favorece a MPI en el sentido de que aparece más a menudo vinculada a super computadores comerciales y de forma mucho más evidente en el caso de los super cluster.

Introducción a 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. 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.



El desarrollo de algoritmos basados en esta arquitectura exige concretar un poco más sus características, principalmente en lo que se refiere a los accesos concurrentes al espacio común:

- EREW: modelo de lectura y escritura exclusivas. Sólo permite que un único procesador lea o escriba una posición de memoria.
- CREW: modelo de lectura concurrente y escritura exclusiva: Permite la escritura por parte de un solo procesador pero varios pueden leer una misma posición de memoria de forma simultánea.
- ERCW: modelo de lectura exclusiva y escritura concurrente. Se trata de la opción opuesta a la anterior.
- CRCW: modelo de lectura y escritura concurrente. No presenta ninguna restricción para la lectura o escritura de las posiciones comunes.

Es evidente que en los casos de escritura concurrente deben existir métodos de gestión de conflictos basados en prioridades u otro tipo de políticas.

A pesar de que un modelo en el que cualquiera puede leer una posición de memoria, pero solamente uno puede escribirla parece lo más lógico, hay que pensar que ha de existir un hardware que lo soporte. Esto se daría en máquinas MISD, habitualmente denominadas arrays sistólicos en las que cada procesador realiza una operación diferente sobre los mismos datos. Estas arquitecturas no constituyen una opción muy extendida. Por el contrario, la situación más habitual es tener una máquina multiprocesador con una memoria común a la que se conectan todos los procesadores mediante un bus común. En este caso habría que suponer un modelo PRAM-CRCW e implementar una correcta gestión de conflictos.

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 un 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.

Lenguajes para programación en paralelo

Para realizar programas en paralelo se han desarrollados una serie de lenguajes que proporcionan al programador las herramientas que le permiten estructurar los programas en procesos y gestionar la comunicación entre ellos. Básicamente existen tres tipos de lenguajes de programación en paralelo:

- Lenguajes específicos creados para este tipo de programación. Occam o Linda son algunos ejemplos.
- Lenguajes extendidos: añaden extensiones a lenguajes tradicionales como Fortran que les permiten trabajar con estructuras paralelas.
- Extensiones: se añaden a lenguajes tradicionales como C o Fortran para proporcionarles la capacidad de tratar con estructuras paralelas. En este caso no hace falta un compilador específico ya que estas extensiones se presentan como librerías de funciones o rutinas.

En este punto vamos a ir centrando nuestros objetivos. De las opciones comentadas, cualquiera de ellas es válida para profundizar en la programación de sistemas paralelos, pero hay que estudiar cual puede ser la más apropiada según las circunstancias. Los lenguajes específicos van a trabajar sobre un hardware también específico en la mayoría de los casos, por lo que si no se dispone de él no resultan una opción viable. Los otros dos casos resultan bastante similares, aunque los lenguajes extendidos también se suelen emplear para determinados tipos de máquinas y no de forma general, se podría destacar otra diferencia interesante. En caso de emplear un lenguaje extendido como por ejemplo Fortran 90, se hace necesario el correspondiente compilador que debe estar disponible para la plataforma en la que se esté trabajando. En caso de emplear extensiones, no se hace necesario ningún cambio en la forma de trabajo habitual, suponiendo que ya se esté trabajando o se conozca un lenguaje como C o Fortran. De esta forma, la opción que nos permite sacar el máximo rendimiento del hardware y software ya disponible, con un mínimo esfuerzo tanto de inversión como de tiempo de aprendizaje es el empleo de extensiones.

La decisión está tomada. Solamente resta decidir cuál de los sistemas existentes, todos ellos basados en paso de mensajes, vamos a utilizar. Hemos analizado dos sistemas

diferentes: PVM (máquina virtual paralela) y MPI (interfaz de paso de mensajes). A efectos prácticos podemos plantearnos diversas consideraciones, a saber:

- ***Estandarización.*** Se considera que PVM es el estándar en programación de cluster de PCs; esto es asumido por quienes trabajan en este campo, ya que ningún organismo de normalización lo ha asumido. No obstante, especialmente en estos casos, el grado de estandarización viene determinado por el grado de implantación. Este dato no está nada claro. No existen estudios fiables, sino opiniones. La impresión con que uno se queda después de investigar en el tema es que MPI es un sistema más dinámico, que se está desarrollando más en la actualidad, pero es sólo una impresión subjetiva.

- ***Prestaciones.*** Se hace necesario valorar la posibilidad de gestión directa de las comunicaciones entre procesos, así como la disponibilidad de funciones potentes que permitan reducir el trabajo de programación de determinadas operaciones. Desde el punto de vista didáctico, es interesante conocer el funcionamiento y la gestión del intercambio de mensajes entre procesos. Desde el punto de vista profesional, resulta más práctico disponer de funciones potentes que reduzcan el coste de programación. En este sentido, MPI permite el trabajo con funciones de gestión directa de las comunicaciones como PVM, pero proporciona funciones no disponibles en PVM que incrementan notablemente la potencia del sistema.

- ***Plataformas.*** Interesante es también analizar en qué plataformas se soportan cada uno de los dos sistemas considerados. En cuanto al sistema operativo, ambas trabajan con UNIX y WINDOWS. PVM pretende trabajar con sistemas WINDOWS 9x, pero ya hemos comprobado que esto no es muy aprovechable, luego en este aspecto no hay muchas diferencias. En cuanto al tipo de máquinas soportadas, la gama es suficientemente amplia en ambos casos como para abarcar las posibilidades de trabajo de la gran mayoría de los programadores, por lo que tampoco debe preocuparnos demasiado este aspecto.

- ***Rendimiento.*** Sería deseable poder obtener el máximo rendimiento en las aplicaciones desarrolladas, ya que este es el objetivo principal de la programación en paralelo. Se han realizado pruebas sobre ambos sistemas y se han consultado datos y opiniones adicionales. La conclusión ha sido que no se

puede saber a ciencia cierta cuál de los dos sistemas resulta intrínsecamente más óptimo.

Finalmente, se basará el desarrollo de las prácticas en MPI, aunque los mismos casos se podrían desarrollar en PVM. Como lenguaje de programación vamos a optar por C, como lenguaje más extendido para todo tipo de aplicaciones y más conocido por la mayoría de los programadores.

A continuación se desarrolla en plan de trabajo.

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.

CONTENIDOS 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.

La primera función MPI será: 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: MPI_Finalize(). 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 (Rank en inglés).

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* 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.

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

int main (int argc,char *argv[])
{
      int mirango,tamano;

      MPI_Init (&argc,&argv);
      MPI_Comm_rank (MPI_COMM_WORLD,&mirango);
      MPI_Comm_size (MPI_COMM_WORLD,&tamano);

      printf ("Proceso %d de %d: Hola Mundo\n",mirango,tamano);

      MPI_Finalize();
      return 0;
}
```

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 Visual C++ 6.0. Los pasos necesarios son:

1. Crear un proyecto como aplicación de consola para Win32.
2. Seleccionar la opción multihilo.
3. Configurar el directorio "include" que proporciona MPI.
4. Configurar el directorio "lib" que proporciona MPI.
5. Configurar las librarías que emplea MPI.
6. Añadir el fichero fuente con el código del programa.
7. Construir la aplicación.

Se dispone de este proceso descrito con más detalle en el apéndice A de este manual.

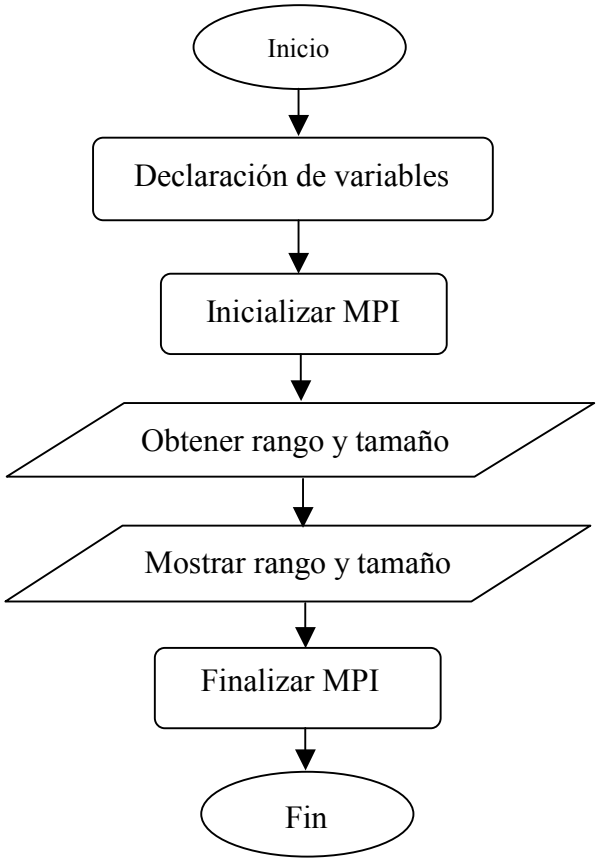
Para ejecutar la aplicación será necesario lanzarla desde el entorno de Argonne National Lab que se encuentra instalado como aplicación en todas las máquinas que forman el cluster. En él nos encontraremos con dos aplicaciones pricipales: “MPD Configuration tool” y “MPIRun”. Vamos a comentar su función. Para más detalles se puede consultar la ayuda que incluyen.

- **MPD Configuration tool** permite realizar algunos ajustes básicos. En este momento conviene recordar que vamos a trabajar en un sistema operativo multiusuario y que un usuario situado en una máquina va a trabajar en varias. No es necesario que se trate de un administrador, pero en caso de no serlo (cosa recomendable sin lugar a dudas), se deben tener en cuenta algunas consideraciones. La cuestión principal es que el usuario esté autorizado a realizar las operaciones que se van a necesitar en todas las máquinas. Si se está trabajando dentro de un dominio, solamente será necesario comprobar que el usuario del dominio disponga de permisos de lectura y escritura en el directorio temporal que configuremos para uso del entorno de trabajo en cada máquina. Si no se ha configurado un dominio para todas las máquinas o éstas pertenecen a distintos dominios, se deben crear usuarios idénticos en todas las máquinas. De esta forma, las máquinas remotas verán al usuario de la máquina local en la cual se lanza la aplicación como usuario propio.
- **MPIRun** es la aplicación del entorno de Argonne que va a lanzar nuestras aplicaciones en las máquinas del cluster que queramos especificar, siempre y cuando se encuentren disponibles. Dispone de varias opciones de funcionamiento, de las cuales vamos a mencionar las más interesantes:
    - **Hosts:** permite configurar qué máquinas formarán parte del cluster.
    - **Advanced options:** establece las opciones de configuración más relevantes. Para nuestro trabajo basta con marcar aquella en la que se señala la exigencia de “password” para lanzar los procesos.
    - **Application:** establece la aplicación que se va a lanzar. Su ruta será la misma en todas las máquinas salvo que se especifique lo contrario en las opciones avanzadas.
    - **Number of proceses:** número de procesos que se desea lanzar.

- **Run:** lanza la aplicación. Pedirá nombre de usuario y password que sean válidos en todas las máquinas configuradas

Siguiendo los pasos descritos se deberá ejecutar la aplicación jugando con las diferentes máquinas disponibles y modificando el número de procesos. Para ejecutarla en máquinas remotas habrá que instalarla en ellas, naturalmente y además con la misma ruta y nombre que en la máquina local; de lo contrario sería necesario especificar exactamente la situación para cada máquina.

Diagrama de flujo del programa



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.

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 estándar, de forma que los hace independientes de la máquina en la que trabaje. De esta forma no es necesario ocuparse de aspectos tales como el formato de coma 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.

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.

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

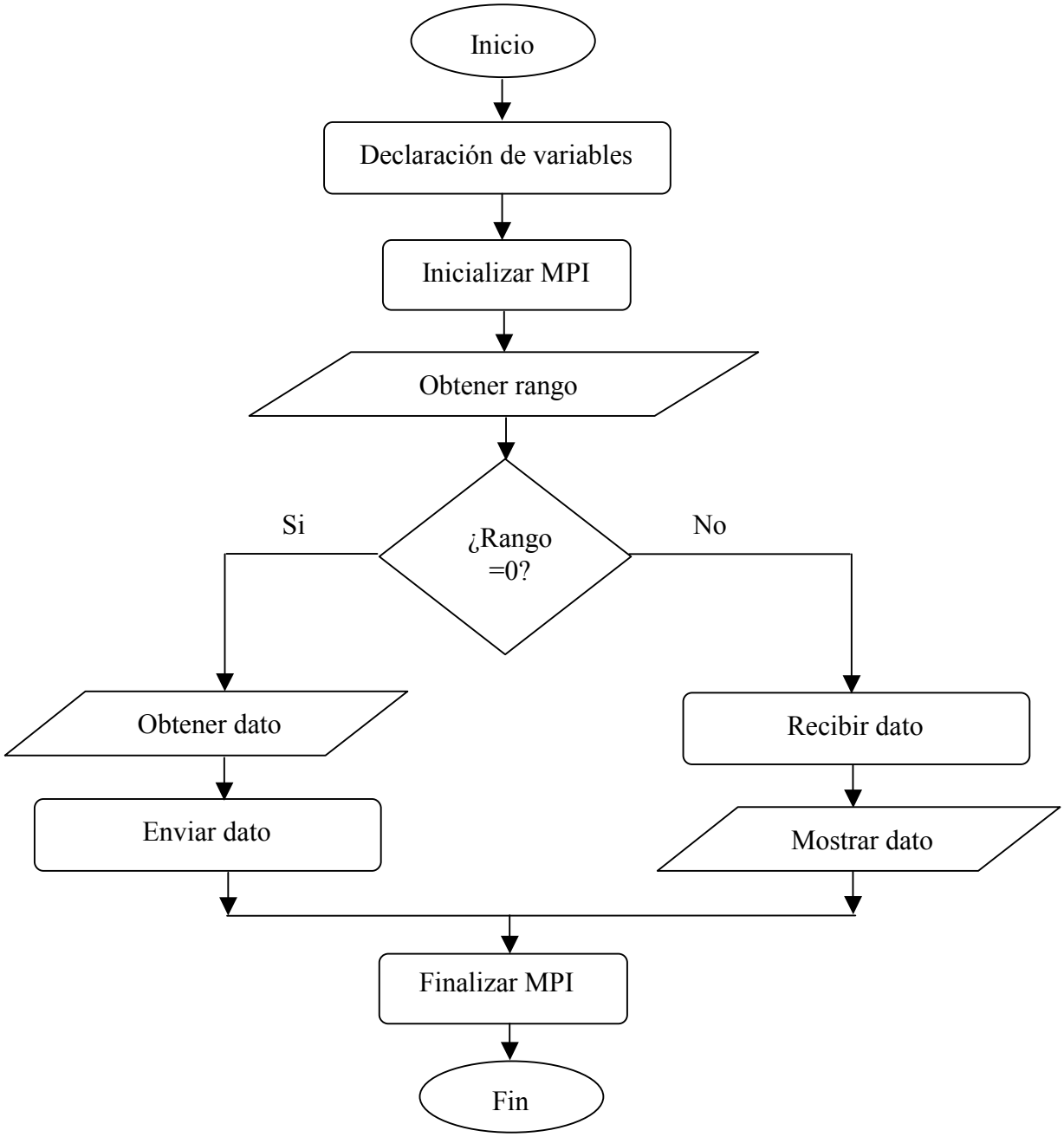
REALIZACIÓN PRÁCTICA

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.

Ejemplo de código que cumple las especificaciones dadas:

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

int main (int argc, char *argv[])
{
      int mirango, tamano;
      MPI_Status estado;
      int dato;

      MPI_Init (&argc,&argv);
      MPI_Comm_rank (MPI_COMM_WORLD,&mirango);

      if (mirango==0)
      {
            printf("Introducir el numero: ");
            fflush(stdout);
            scanf("%d",&dato);
            MPI_Send (dato,1,MPI_INT,1,10,MPI_COMM_WORLD);
      }

      else if (mirango==1)
      {
            MPI_Recv (dato,1,MPI_INT,0,10,MPI_COMM_WORLD,&estado);
            printf (“Me ha llegado el numero: %d \n”,dato);
      }

      MPI_Finalize();
      return 0;
}
```

Diagrama de flujo del programa



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...?

PRÁCTICA 2: COMUNICACIONES COLECTIVAS

OBJETIVOS

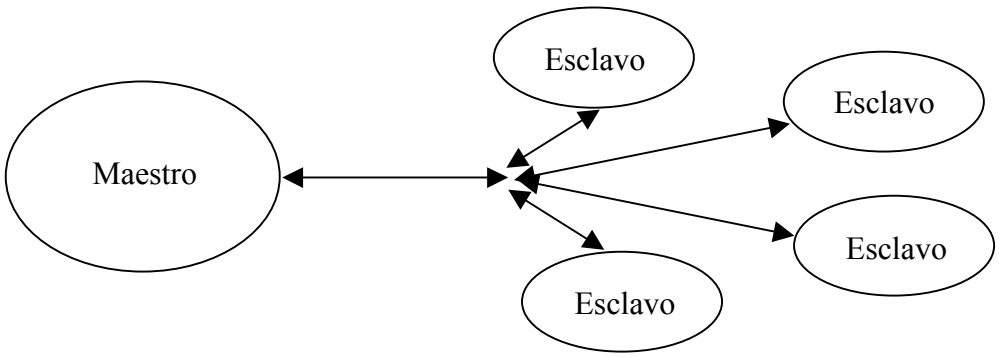
- Extender las posibilidades de comunicación entre procesos a las comunicaciones colectivas como método de simplificación de la comunicación.
- Estudiar posibles aplicaciones de la comunicaciones colectivas.

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.



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 sóla llamada. Se trata de la función: *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. Esta función se emplearía por igual en el código de todos los procesos, pero solamente aquel cuyo rango coincida con "*source*" va a enviar; el resto entienden que deben recoger lo que aquel envíe.

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 *buf, int count, MPI_Datatype dtype, void *buf, int count, MPI_Datatype dtype, int dest, MPI_Comm comm)* y *int MPI_Allgather (void *buf, int count, MPI_Datatype dtype, void *buf, int count, MPI_Datatype dtype, MPI_Comm comm)*. En el primer caso, los datos proporcionados por todos los procesos se almacenan en el buffer de aquel cuyo rango coincida con "*dest*". En el segundo caso los datos se copian en buffers de todos los procesos por igual. Por este motivo no existe argumento de destino.

REALIZACIÓN PRÁCTICA

Como aplicación de los conceptos estudiados vamos a desarrollar un programa de suma de matrices (NxN). 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.

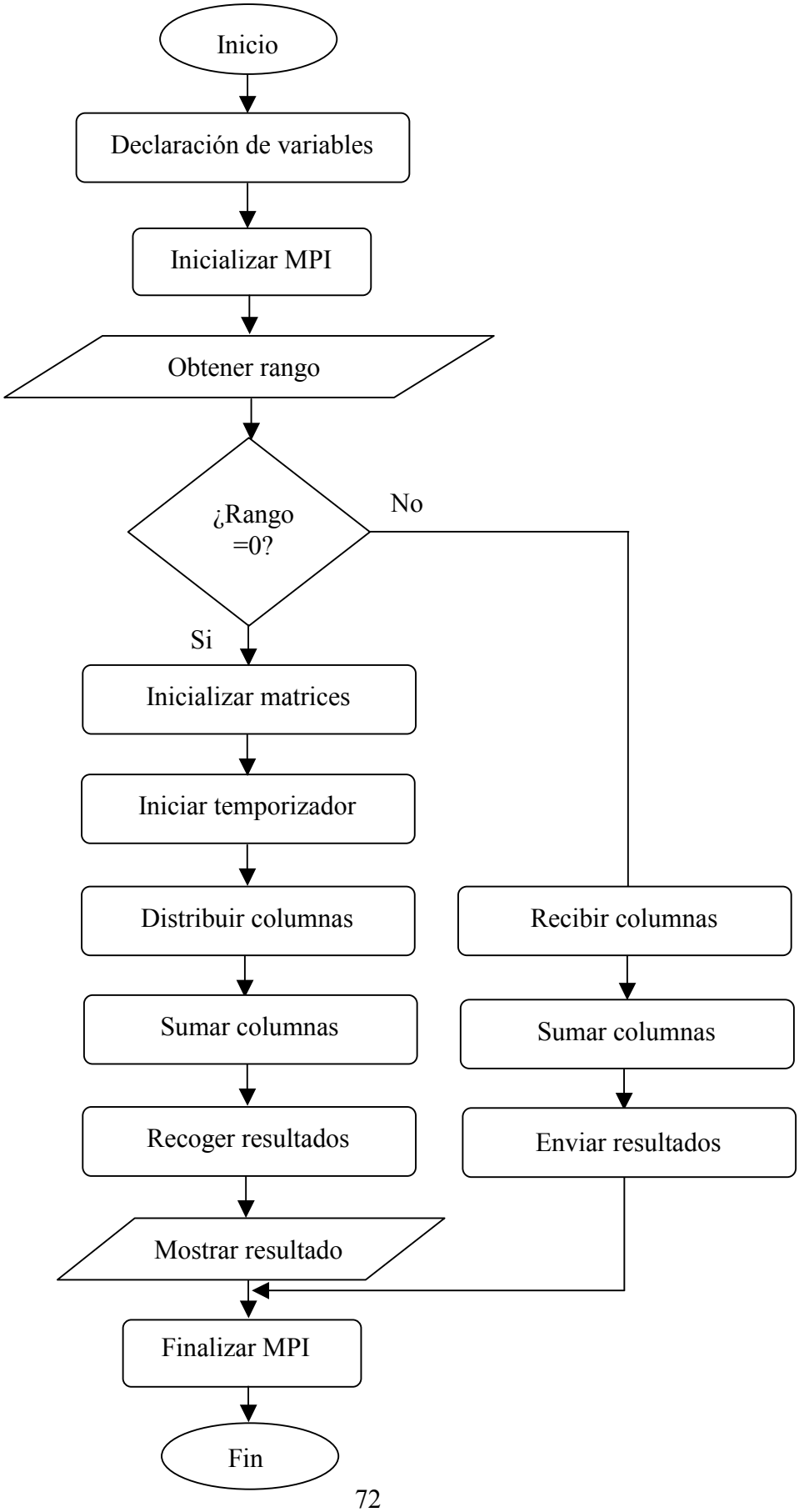
Vamos a añadir la posibilidad de medida del rendimiento de la aplicación. Para ello, MPI proporciona un reloj que nos permite medir la duración del cálculo. Se implementa mediante la función:

double MPI_Wtime(void) que proporciona el tiempo en el instante actual en segundos.

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

#define N 10

int main (int argc,char *argv[])
{
      int mirango,i,j,numero_datos,matriz1[N][N];
      int matriz2[N][N],suma[N],resultado[N*N];
      double tiempo_on,tiempo_off;

      numero_datos=N*N;

      MPI_Init (&argc,&argv);
      MPI_Comm_rank (MPI_COMM_WORLD,&mirango);


      if(mirango==0)
      {
            for(i=0;i<N;i++)
            {
                  for(j=0;j<N;j++)
                  {
                        matriz1[j][i]=matriz2[j][i]=j;
                  }
            }
      }

      if (mirango==0)
      {
            tiempo_on = MPI_Wtime();
      }

/*Tiene que haber N procesos*/

     MPI_Bcast(matriz1,numero_datos,MPI_INT,0,MPI_COMM_WORLD);
     MPI_Bcast(matriz2,numero_datos,MPI_INT,0,MPI_COMM_WORLD);

      for(i=0;i<N;i++)
      {
            suma[i]=matriz1[i][mirango]+matriz1[i][mirango];
      }

      MPI_Gather(suma,N,MPI_INT,resultado,N,MPI_INT,0,MPI_COMM_WORLD);

      if(mirango==0)
      {
            tiempo_off = MPI_Wtime();
            printf("tiempo empleado = %f\n",tiempo_off-tiempo_on);

            for(i=0;i<N;i++)
            {
                  for(j=0;j<N;j++)
                        printf("%d ",resultado[j+i*N]);
                  printf("\n");
            }
      }

      MPI_Finalize();
```

```
        return 0;
}
```

Diagrama de flujo del programa



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.

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 *busendf, 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, los datos contenidos en el buffer del proceso emisor son copiados en los buffers de los procesos receptores. El primer parámetro "*count*" indica cuántos datos van a cada proceso.

La función de reducción es: *int MPI_Reduce (void *bufsend, void *bufrecv, int count, MPI_Datatype dtype,MPI_op op, int dest, MPI_Comm comm).* Ante esta función, todos los procesos envían sus datos a aquel cuyo rango coincide con el parámetro "*dest*", el cual no los almacena en su buffer (segundo puntero *buf*), sino que lo que ahí coloca es el resultado de realizar sobre estos datos la operación indicada en "*op*". Este parámetro es

del tipo "*MPI_op*", es decir, una operación MPI. Las operaciones MPI son las siguientes:

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 lógico
MPI_MINLOC	determina el rango del proceso que contiene el valor menor.
MPI_MAXLOC	determina e rango del proceso que contiene el valor mayor.

REALIZACIÓN PRÁCTICA

En esta práctica se va a programar el cálculo del número "e" a partir de la descomposición de la función e^x en serie de potencias en torno al origen. Tenemos que:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + ... + \frac{x^n}{n!}$$

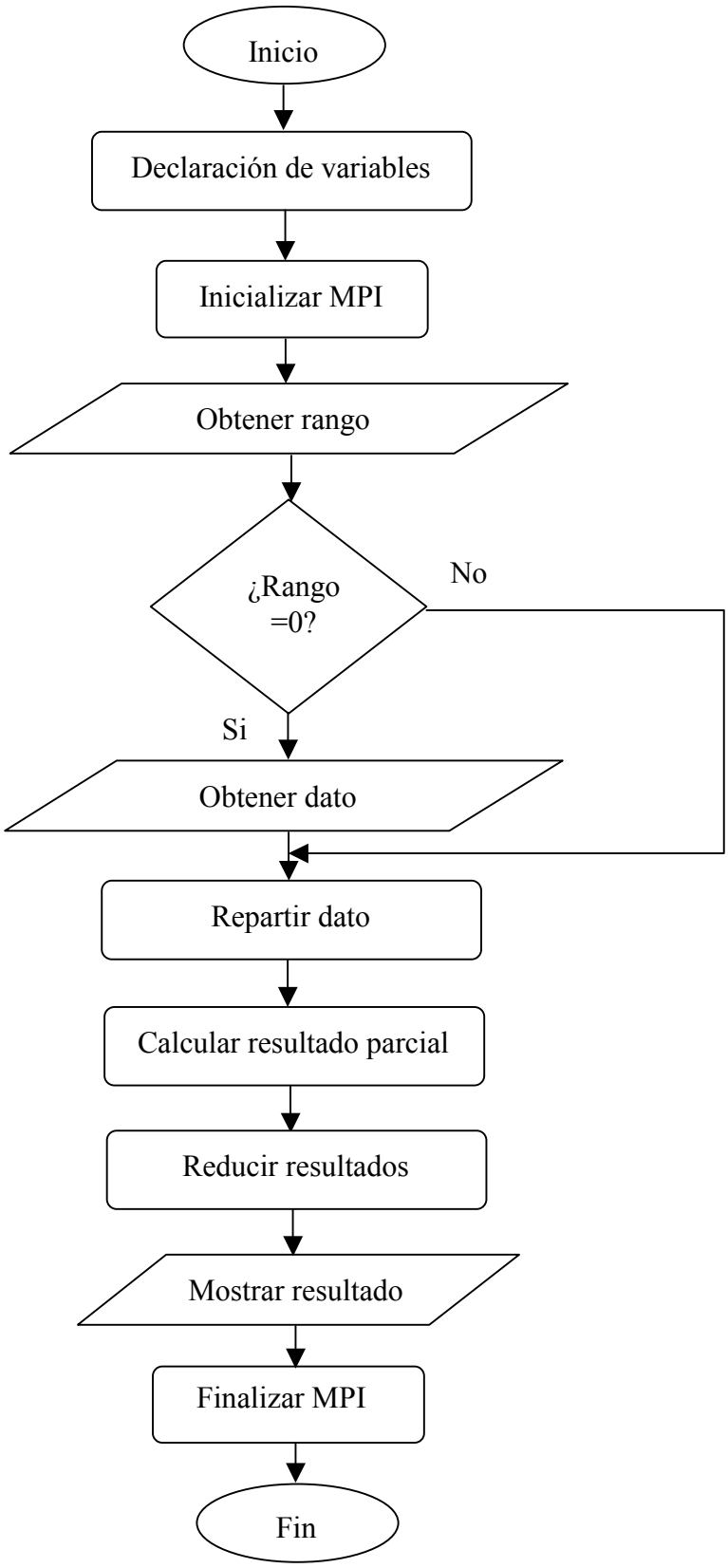
Se propone construir la aplicación de manera que cada proceso calcule un término de las serie. El proceso 0 proporciona la x (que valdrá 1 si se quiere calcular e) introducida por el usuario y muestra el resultado. Obviamente, cuantos más procesos se arranquen, más preciso será el cálculo.

Ejemplo de código que cumple las especificaciones dadas:

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

int main (int argc, char *argv[])
{
      int mirango,i;
      float x,resultado,miresultado,factorial;

      MPI_Init (&argc,&argv);
      MPI_Comm_rank (MPI_COMM_WORLD,&mirango);

      if (mirango == 0)
      {
```

```
            printf("Introducir x: ");fflush(stdout);
            scanf("%f",&x);
      }
      MPI_Bcast(&x,1,MPI_FLOAT,0,MPI_COMM_WORLD);

      miresultado=powf(x,mirango);
      factorial=1;
      for(i=1;i<=mirango;i++)
      {
            factorial=factorial*i;
      }

      miresultado=miresultado/factorial;

      MPI_Reduce(&miresultado,&resultado,1,MPI_FLOAT,MPI_SUM,0,
                  MPI_COMM_WORLD);

      if (mirango==0)
      {
            printf("el resultado es: %.16f\n",resultado);
      }
      MPI_Finalize();
      return 0;
}
```

Diagrama de flujo del programa



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?

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 van 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 *cmm1* empleando para ello coordenadas cartesianas e incluyéndolos en *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, crear un vector de dos enteros, 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 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 n filas y m columnas. Esto asignaría coordenadas a NxM procesos. En caso, por ejemplo, de declarar periódicas las filas, cualquier referencia a la fila n, se asimilaría de nuevo a la fila 0; una referencia la fila n+1 se entendería como una referencia a la fila n+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 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:

MPI_Cart_coord (MPI_Comm comm, int rank, int ndims, int cords)*. Esta función devuelve las coordenadas cartesianas del proceso con rango *rank* en el vector apuntado por *cords*, cuya dimensión debe coincidir con *ndims* y éste a su vez con el número de dimensiones de la matriz de procesos.

REALIZACIÓN PRÁCTICA

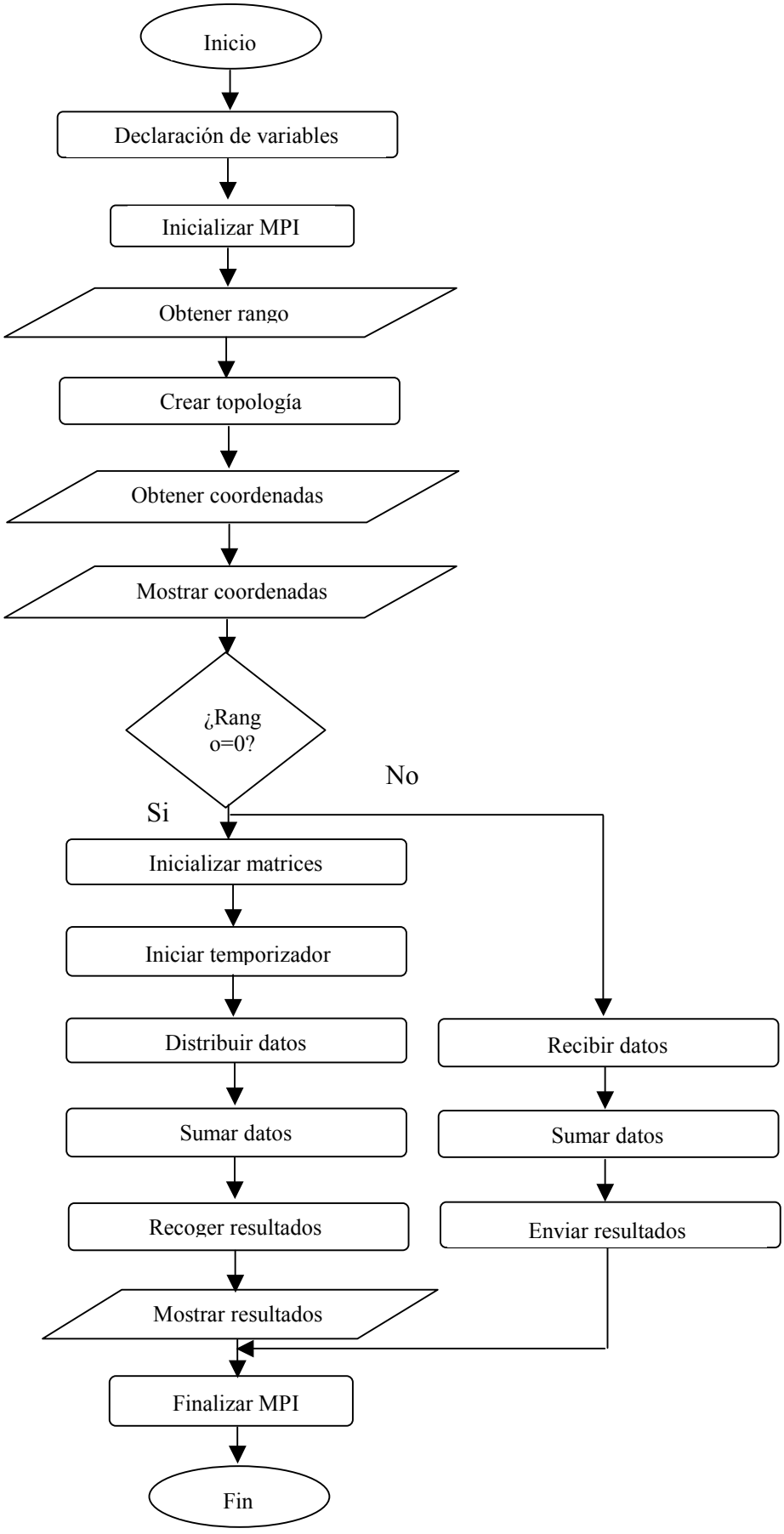
Se reprogramará la suma de matrices de manera que cada proceso dentro de una topología cartesiana sume uno de los elementos de cada matriz. El resultado se mostrará, junto con el tiempo empleado de igual manera que en la práctica 2.

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

#define N 3
```

```
int main (int argc,char *argv[])
{
      int mirango,i,j,numero_datos,matriz1[N][N];
      int matriz2[N][N],suma,resultado[N*N];
      double tiempo_on,tiempo_off;

      int ndims,reorder,periodos[2],dim_size[2],coords[2];
      MPI_Comm ant_com,nue_com;
      ant_com=MPI_COMM_WORLD;

      ndims=2;
      dim_size[0]=3;
      dim_size[1]=3;
      periodos[0]=1;
      periodos[1]=1;
      reorder=1;
      coords[0]=0;
      coords[1]=0;
      numero_datos=N*N;

      MPI_Init (&argc,&argv);
      MPI_Comm_rank (MPI_COMM_WORLD,&mirango);

      MPI_Cart_create (ant_com,ndims,dim_size,periodos,reorder,
                       &nue_com);

      MPI_Cart_coords(nue_com,mirango,2,coords);
      printf("%d, %d,%d\n",mirango,coords[0],coords[1]);

      if(mirango==0)
      {
            for(i=0;i<N;i++)
            {
                  for(j=0;j<N;j++)
                  {
                        matriz1[i][j]=matriz2[i][j]=j;
                  }
            }
      }

      if (mirango==0)
      {
            tiempo_on = MPI_Wtime();
      }

/*Tiene que haber N*N procesos*/

      MPI_Bcast(matriz1, numero_datos, MPI_INT, 0, MPI_COMM_WORLD);
      MPI_Bcast(matriz2, numero_datos, MPI_INT, 0, MPI_COMM_WORLD);


     suma=matriz1[coords[0]][coords[1]]+matriz1[coords[0]][coords[1]];

      MPI_Gather(&suma,1,MPI_INT,resultado,1,MPI_INT,0,
                 MPI_COMM_WORLD);

      if(mirango==0)
      {
            tiempo_off = MPI_Wtime();
            printf("tiempo empleado = %f\n",tiempo_off-tiempo_on);
```

```
            for(i=0;i<N;i++)
            {
                  for(j=0;j<N;j++)
                        printf("%d ",resultado[j+i*N]);
                  printf("\n");
            }
      }

      MPI_Finalize();
      return 0;
}
```

Diagrama de flujo del programa



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.

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 puede 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 tendrá su propia zona de fichero para sus datos; 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, escritura, etc; cerrarlo. 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 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 imlementación de MPI. Se suele pasar como nulo (MPI_INFO_NULL) dad su escasa utilidad. Finalmente se debe especificar que los modos de acceso al fichero ueden ser los siguientes:

MPI_MODE_RONLY	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

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 vana intercalar los datos introducidos por los diferentes procesos. Este parámetro proporciona una notable potencia a la esntrada/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 nativos 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 “external” que solamente realiza la conversión de datos si es necesaria para que estos sean entendidos por todas las máquinas.

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:

*MPI_File_close(MPI_File *fh)*

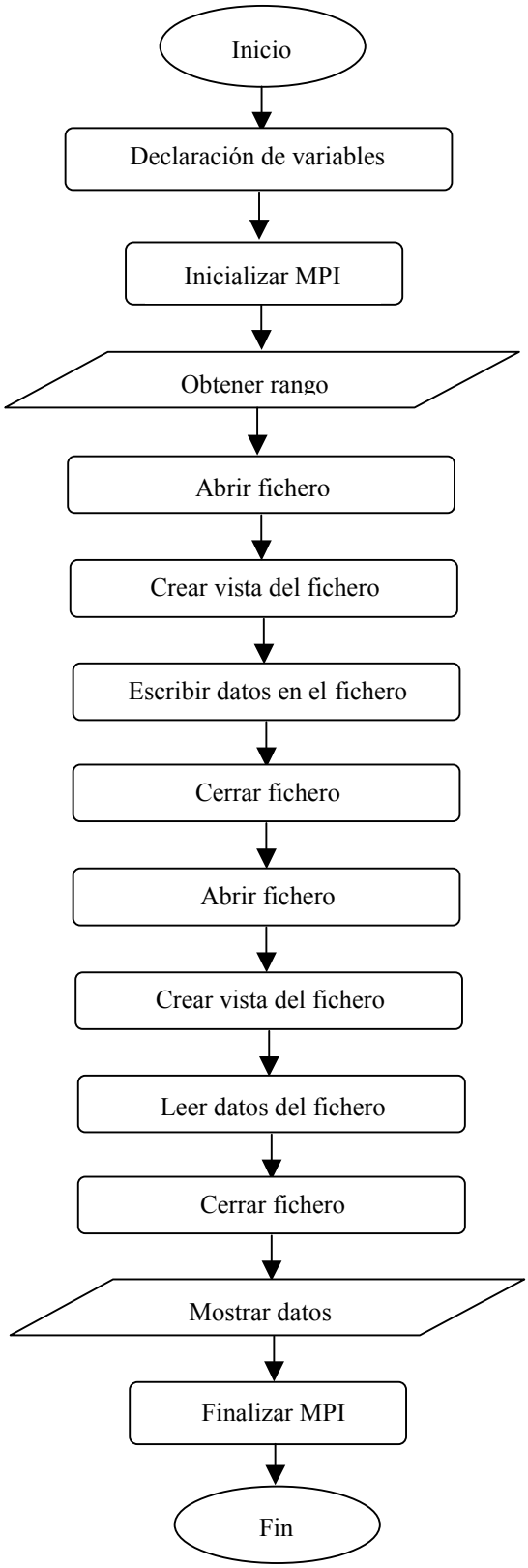
REALIZACIÓN PRÁCTICA

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), a continuación uno de otro y por orden de rango. Posteriormente se 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.

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

#define ndatos 20

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

      int escribir[ndatos], leer[ndatos], i, mirango;
      MPI_File fh;
      MPI_Status estado;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &mirango);

      for (i=0;i<ndatos;i++)
      {
            escribir[i]=48+mirango;
      }

      MPI_File_open(MPI_COMM_WORLD, "archivo.dat",MPI_MODE_CREATE |
                  MPI_MODE_RDWR,MPI_INFO_NULL, &fh);
      MPI_File_set_view(fh, ndatos*mirango*sizeof(int), MPI_INT,
                        MPI_INT, "native",MPI_INFO_NULL);
      MPI_File_write_at(fh, 0, escribir, ndatos, MPI_INT, &estado);
      MPI_File_close(&fh);

      MPI_File_open(MPI_COMM_WORLD, "archivo.dat",MPI_MODE_CREATE |
                  MPI_MODE_RDWR,MPI_INFO_NULL, &fh);
      MPI_File_set_view(fh, ndatos*mirango*sizeof(int), MPI_INT,
                        MPI_INT, "native",MPI_INFO_NULL);
      MPI_File_read_at(fh, 0, leer, ndatos, MPI_INT, &estado);
      MPI_File_close(&fh);

      for(i=0;i<ndatos;i++)
      {
            leer[i]=leer[i]-48;
      }

      printf("Proceso %d\n ", mirango);
```

```
	for (i=0; i<ndatos; i++)
	{
		printf("%d ", leer[i]);
	}
	printf("\n");

	MPI_Finalize();
	return 0;
}
```

Diagrama de flujo del programa



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?

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 que 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 que indica si es "1" que la operación ha terminado; de lo contrario devuelve un "0".

REALIZACIÓN PRÁCTICA

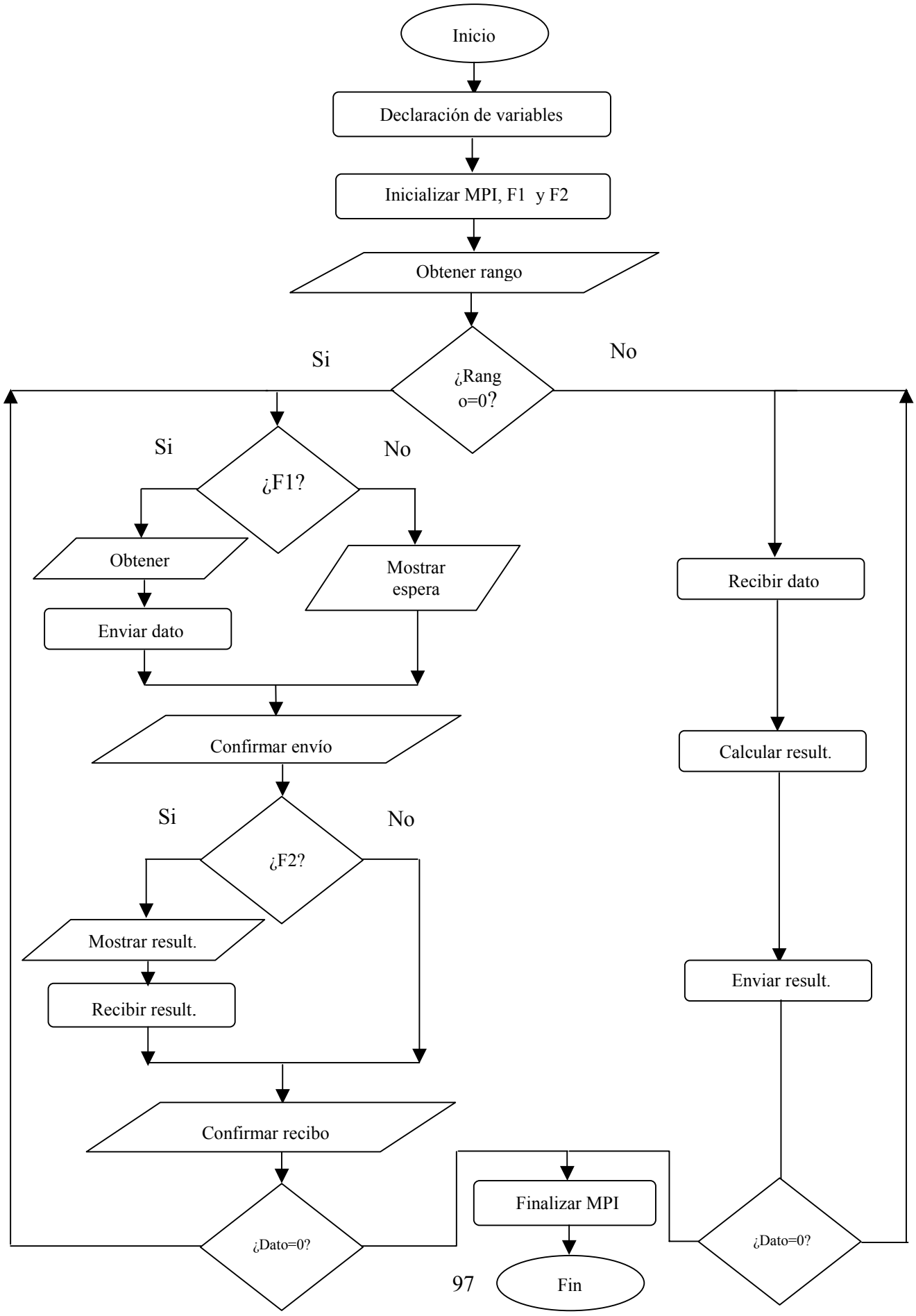
Vamos a suponer el siguiente escenario: el proceso cero se encarga de calcular el factorial de los números que le va introduciendo el usuario. El cálculo se lo encarga al proceso 1 que se supone que reside en una máquina con mayor potencia de cálculo. Para evitar el colapso de las peticiones, deberá 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 incluiremos por seguridad y como forma de aprovechar la potencia de las nuevas funciones. Se puede comprobar su funcionamiento repitiendo el cálculo un determinado número de veces para forzar la espera del proceso cero.

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

```
int main (int argc,char *argv[])
{
      int mirango,i;
      MPI_Status estado;
      int dato[1];
      float factorial1,factorial2;
      MPI_Request request1,request2;
      int testigo1,testigo2;

      testigo1=1;
      testigo2=1;
      factorial2=0;

      MPI_Init (&argc,&argv);
      MPI_Comm_rank (MPI_COMM_WORLD,&mirango);

      if (mirango==0)
      {
            do
            {
                  if(testigo1==1)
                  {
                        printf("Introducir el numero: ");
                        fflush(stdout);
                        scanf("%d",&dato);
                        MPI_Isend (dato,1,MPI_INT,1,10,
                                    MPI_COMM_WORLD,&request1);
                  }
                  else
                  {
                        printf("espere.....\n");
                  }
                  MPI_Test (&request1,&testigo1,&estado);

                  if(testigo2==1)
                  {
                        printf("El resultado es: %f\n",factorial2);
                        MPI_Irecv (&factorial2,1,MPI_FLOAT,
                                    1,10,MPI_COMM_WORLD,&request2);
                  }
                  MPI_Test (&request2,&testigo2,&estado);

            }while (*dato>0);
      }

      else if (mirango==1)
      {
            do
            {
                  MPI_Recv (dato,1,MPI_INT,0,10, MPI_COMM_WORLD,
                            &estado);
                  factorial1=1;
                  for(i=1;i<=*dato;i++)
                  {
                        factorial1=factorial1*i;
                  }
                  MPI_Send (&factorial1,1,MPI_FLOAT,0,10,
                            MPI_COMM_WORLD);
            }while (*dato>0);
      }
      MPI_Finalize();
```

```
        return 0;
}
```

Diagrama de flujo del programa



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.

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 elementos del tipo derivado. El parámetro *vector_longit_bloques* contiene en número de datos de cada elemento. El parámetro *vector_desplazam* especifica el desplazamiento de cada elemento respecto del inicio del mensaje. El parámetro *vector_tipos* asigna a cada elemento un tipo de datos definido en MPI. 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 *nue_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 <stdio.h>
#include <math.h>
#include <mpi.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 a;
            int b;
            char c;
      }viejo_tipo;

      viejo_tipo datos;

      vector_long[0]=vector_long[1]=vector_long[2]=1;

      vector_despl[0]=0;
      vector_despl[1]=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.a=1.5;
            datos.b=10;
            datos.c='2';
      }
      MPI_Bcast(&datos, 1, nuevo_tipo, 0, MPI_COMM_WORLD);

      if(mirango!=0)
      {
            printf("a=%f b=%d c=%c",datos.a,datos.b,datos.c);
      }
      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 hace lo mismo pero tomando *long_bloque* elementos consecutivos cada *salto* elementos, repitiendo la operación *count* veces.

La tercera toma *count* elementos, de forma que el enésimo elemento de ellos es un vector cuyo número de componentes viene dado por la enésima componente de *vector_longitudes* y son del tipo *tipo_elemento*. Este vector se encuentra desplazado respecto del inicio del nuevo tipo el tamaño de *tipo_elemento* multiplicado por la

enésima componente de *vector_desplazamiento*. Resulta un poco complicado pero puede proporcionar mucha potencia en determinados casos.

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 derivados que se acaban de ver vamos replantear la aplicación de suma de matrices para hacerla más potente y flexible. Se tratará de que cada proceso arrancado calcule la suma de una submatriz cuyo tamaño será calculado en función de los procesos en que quiera el usuario dividir la dimensión horizontal de las matrices, ajustando la división de la dimensión vertical al número de procesos disponibles. Por ejemplo, si tenemos una matriz 10x10, lanzamos 4 procesos y el usuario decide que habrá dos en la dimensión horizontal, cada proceso calculará matrices de tamaño mitad que la original tanto en filas como en columnas, es decir, 4 matrices 5x5. Este es un caso muy simple, pero si la división no es exacta, por ejemplo con matrices 11x11, se deberá prever la situación para que se pueda seguir calculando.

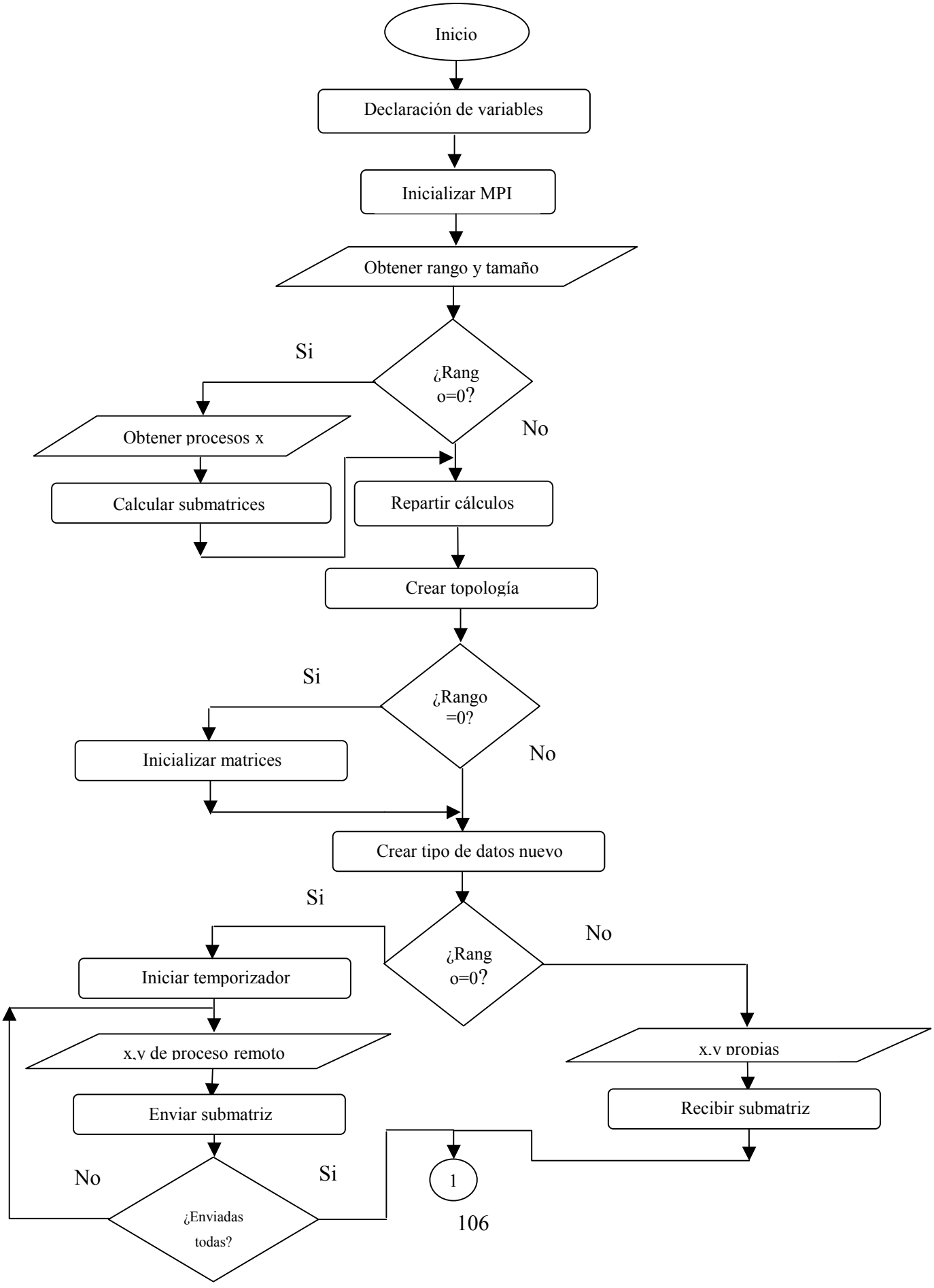
```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <mpi.h>

#define N 10
#define M 10

int main (int argc, char *argv[])
{
      int mirango,procesos,proc_fila,proc_col,submatriz_f,submatriz_c;
      int i,j,matriz1[2*M][2*N],matriz2[2*M][2*N],suma[2*M][2*N];
      double tiempo_on,tiempo_off;
      int ndims, reorder, periodos[2], dim_size[2], coords[2];
      MPI_Comm ant_com, nue_com;
      void *origen1,*origen2;

      ant_com=MPI_COMM_WORLD;

      MPI_Init (&argc, &argv);
      MPI_Comm_rank (MPI_COMM_WORLD, &mirango);
      MPI_Comm_size (MPI_COMM_WORLD, &procesos);

      if (mirango == 0)
      {
            printf("Procesos en eje x: ");
            fflush(stdout);
```

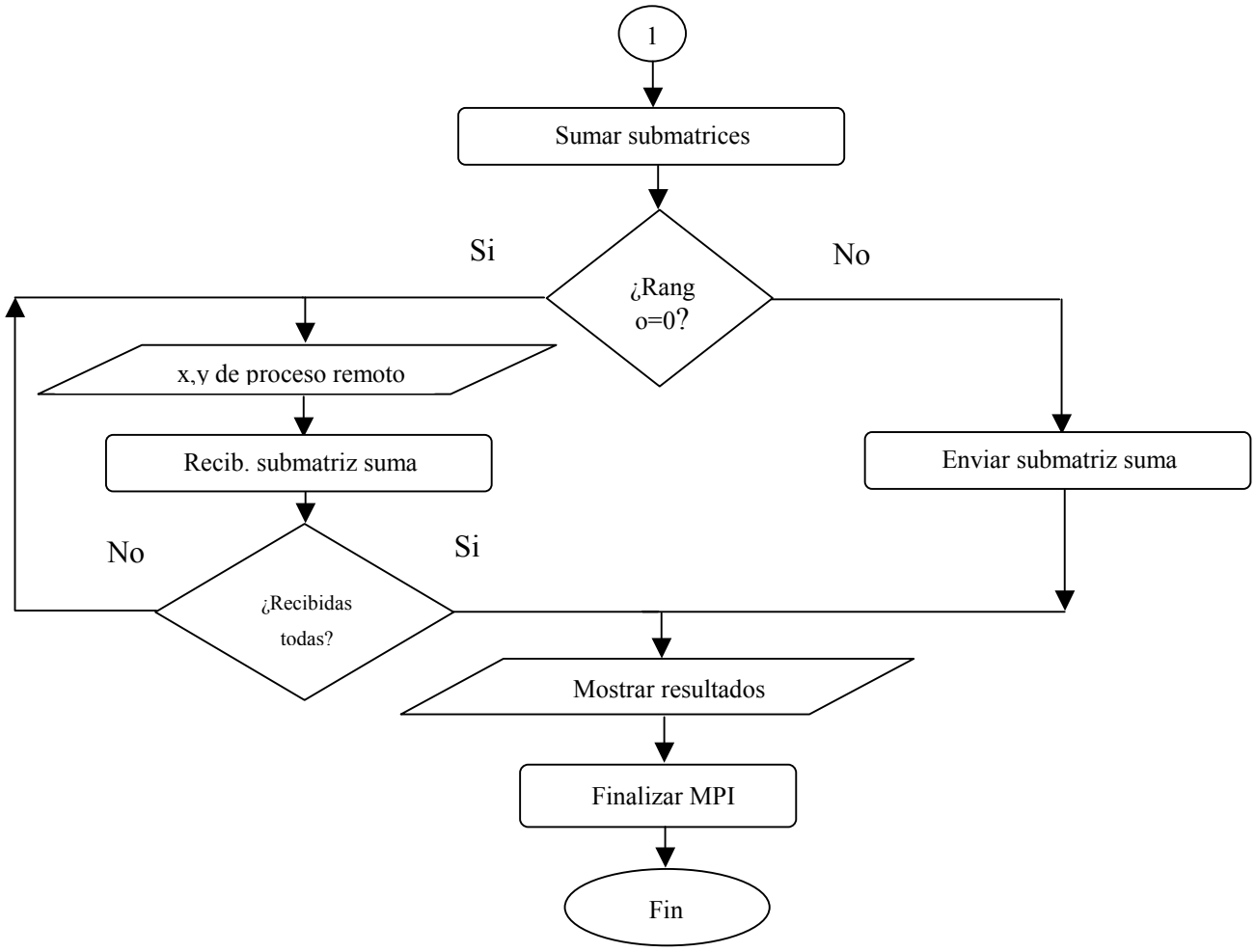
```
	scanf("%d",&proc_col);
	proc_fila=procesos/proc_col;
	submatriz_f=M/proc_fila;
	if(M>(submatriz_f*proc_fila))
		submatriz_f++;
	submatriz_c=N/proc_col;
	if(N>(submatriz_c*proc_col))
		submatriz_c++;
}

MPI_Bcast(&submatriz_f, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&submatriz_c, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&proc_fila, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&proc_col, 1, MPI_INT, 0, MPI_COMM_WORLD);

ndims=2;
dim_size[0]=proc_fila;
dim_size[1]=proc_col;
periodos[0]=1;
periodos[1]=1;
reorder=1;
coords[0]=0;
coords[1]=0;

MPI_Cart_create (ant_com, ndims, dim_size, periodos, reorder,
		&nue_com);

if(mirango==0)
{
	for(i=0;i<M;i++)
	{
		for(j=0;j<N;j++)
		{
			matriz1[i][j]=matriz2[i][j]=i;
		}
	}
}

int vector_long[M];
MPI_Aint vector_despl[M];
MPI_Datatype vector_tipos[M];
MPI_Datatype nuevo_tipo;
MPI_Status estado;

vector_long[0]=submatriz_c;
vector_despl[0]=0;
vector_tipos[0]=MPI_INT;
for(i=1;i<submatriz_f;i++)
{
	vector_long[i]=submatriz_c;
	vector_despl[i]=vector_despl[i-1]+2*N*sizeof(int);
	vector_tipos[i]=MPI_INT;
}

MPI_Type_struct(submatriz_f,vector_long,vector_despl,
		vector_tipos,&nuevo_tipo);
MPI_Type_commit(&nuevo_tipo);


if (mirango==0)
{
```

```
        tiempo_on = MPI_Wtime();

        for(i=1;i<procesos;i++)
        {
                MPI_Cart_coords(nue_com, i, 2, coords);
                origen1=&matriz1[coords[0]*submatriz_f]
                        [coords[1]*submatriz_c];
                origen2=&matriz2[coords[0]*submatriz_f]
                        [coords[1]*submatriz_c];
                MPI_Send(origen1,1,nuevo_tipo,i,10,MPI_COMM_WORLD);
                MPI_Send(origen2,1,nuevo_tipo,i,10,MPI_COMM_WORLD);
        }
        MPI_Cart_coords(nue_com, 0, 2, coords);
}

if (mirango!=0)
{
        MPI_Cart_coords(nue_com, mirango, 2, coords);
        origen1=&matriz1[coords[0]*submatriz_f]
                [coords[1]*submatriz_c];
        origen2=&matriz2[coords[0]*submatriz_f]
                [coords[1]*submatriz_c];
        MPI_Recv(origen1,1,nuevo_tipo,0,10,MPI_COMM_WORLD,&estado);
        MPI_Recv(origen2,1,nuevo_tipo,0,10,MPI_COMM_WORLD,&estado);
}


for(i=coords[0]*submatriz_f;i<((coords[0]+1)*submatriz_f);i++)
{

for(j=coords[1]*submatriz_c;j<((coords[1]+1)*submatriz_c);j++)
        {
                suma[i][j]=matriz1[i][j]+matriz2[i][j];
        }
}

if(mirango!=0)
{
        origen1=&suma[coords[0]*submatriz_f]
                [coords[1]*submatriz_c];
        MPI_Send(origen1,1,nuevo_tipo,0,10,MPI_COMM_WORLD);
}


if(mirango==0)
{
        for(i=1;i<procesos;i++)
        {
                MPI_Cart_coords(nue_com, i, 2, coords);
                origen1=&suma[coords[0]*submatriz_f]
                        [coords[1]*submatriz_c];
                MPI_Recv(origen1,1,nuevo_tipo,i,10,MPI_COMM_WORLD,
                        &estado);
        }
}

if(mirango==0)
{
        tiempo_off = MPI_Wtime();
        printf("tiempo empleado = %f\n",tiempo_off-tiempo_on);
```

```
            for(i=0;i<M;i++)
            {
                  for(j=0;j<N;j++)
                        printf("%d ",suma[i][j]);
                  printf("\n");
            }
      }

      MPI_Finalize();
      return 0;
}
```

Diagrama de flujo del programa




1
Sumar submatrices
Si
¿Rang
o=0?
No
x,y de proceso remoto
Recib. submatriz suma
Enviar submatriz suma
No
¿Recibidas
todas?
Si
Mostrar resultados
Finalizar MPI
Fin

CUESTIONES

- En la práctica realizada, ¿qué problemas aparecen si se realiza reserva dinámica de memoria para crear especio 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.

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 maxprocs, MPI_Info info, int root, MPI_Comm comm., MPI_Comm *intercom, int array_of_errcodes[])*

Esta función intentará lanzar *maxprocs* copias del programa MPI apuntado por *command*, devolviendo el intercomunicador apuntado por *intercomm* para el intercambio de información con ellos. También es posible la comunicación entre los procesos hijos ya que un nuevo MPI_COMM_WORLD se crea automáticamente incluyendo exclusivamente a los nuevos procesos. Mediante *argv* se pueden pasar argumentos al programa lanzado. El parámetro *root* es la identificación del proceso padre dentro del comunicador *comm*. Mediante el parámetro *info*, se proporcionan una serie de valores acerca de dónde y cómo arrancar los procesos hijos. Finalmente, se devuelve un *array* de posibles códigos de error, uno por proceso lanzado.

Para una fácil utilización de la función descrita, se puede pasar como comando el nombre del programa hijo directamente entrecomillado; como info se puede pasar MPI_INFO_NULL y como array MPI_ERRORCODES_IGNORE. Si no existen argumentos en línea de comandos para el programa hijo, se puede pasar MPI_ARGV_NULL.

Otra función útil en la gestión de procesos es:

*int MPI_Comm_get_parent(MPI_Comm *comm)*

Devuelve a los procesos hijos el intercomunicador que les permite intercambiar información con el proceso padre.

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

Permite conocer a través del puntero *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. Se puede pasar el contenido del puntero anterior como parámetro m*axprocs* a la función *MPI_Comm_spawn*. Si esta funcionalidad no está soportada el contenido de *flag* será falso.

Nota importante: probablemente debido a la novedad de las funciones de gestión dinámica del cluster, se han detectado algunos errores en su implementación en LAM-MPI que apenas de encuentran reflejados en la lista de mensajes de la página web de LAM en la fecha de redacción de este manual. El más importante de ellos es el que se produce al intentar transferir información del proceso "padre" a los "hijos" empleando el intercomunicador proporcionado por la función *MPI_Comm_spawn*. Al parecer, éste adquiere distinto valor en el padre y en los hijos, por lo que es imposible la comunicación. La solución está en transformar a ambos lados el intercomunicador en intracomunicador mediante la función:

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

Se emplea para crear un intracomunicador agrupando a los grupos de procesos comunicados mediante un intercomunicador. El parámetro *orden* indica cómo se reordenaran los rangos de los procesos en el nuevo grupo. Así, pondremos este parámetro a falso (0) por ejemplo en el padre y a verdadero (1) en los hijos para que el padre tenga en rango cero en el nuevo intracomunicador. Los hijos quedarán ordenados como estaban en su propio intracomunicador, pero comenzando por el rango 1.

Otro error encontrado se produce solamente al transferir información de tipo string entre procesos hijos, no entre el padre y los hijos. Se produce un error si el puntero correspondiente tiene el mismo nombre en el emisor y el receptor. Solamente ocurre con strings; no sucede con ningún otro tipo de variable.

REALIZACIÓN PRÁCTICA

Como práctica de los conceptos que acabamos de ver se va a realizar un sencillo programa en el que un único proceso lanzado inicialmente va a arrancar procesos hijos en todos los nodos disponibles. Se mandarán mensajes de saludo: del padre a los hijos y del hijo de menor rango al resto. Para comprobar el funcionamiento, los hijos imprimirán el mensaje de saludo del padre, el suyo propio y un mensaje de saludo al padre que no llegarán a enviar.

```
/*Programa padre.c*/
#include <mpi.h>

int main (int argc, char *argv[])
{
      int tamano,universe_size;
      int *universe_sizep,flag;
      char *saludo_padre;

      MPI_Comm intercom,intracom;

      saludo_padre=”Hola”;

      MPI_INIT(&argc,&argv);
      MPI_Attr_get(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE, &universe_sizep,
                  &flag);
      if(!flag)
      {
            printf(“UNIVERSE_SIZE no soportado. Cuantos hijos?”);
            scanf(“%d”,&universe_size);
      }
      else universe_size=*universe_sizep;

      MPI_Comm_spawn(“hijo”,MPI_ARGV_NULL,universe_size,MPI_INFO_NULL,
                  0,MPI_COMM_WORLD,&intercom,MPI_ERRCODES_IGNORE);
      MPI_Intercomm_merge(intercom,0,&intracom);
      MPI_Bcast(saludo_padre,4,MPI_CHAR,0,intracom);
      MPI_Finalize();
      return 0;
}

/*Programa hijo.c*/
#include <mpi.h>

int main (int argc, char *argv[])
{
      int tamano,mirango;
      char *saludo_padre;
      char *saludo_hijo;
      char *saludo_her;

      MPI_Comm parent,intracom;

      MPI_INIT(&argc,&argv);
      MPI_Comm_get_parent(&parent);
```

```
	MPI_Intercomm_merge(parent,1,&intracom);
	MPI_Bcast(saludo_padre,4,MPI_CHAR,0,intracom);
	printf(“%.4s\n”,saludo_padre);
	MPI_Comm_rank(MPI_COMM_WORLD,&mirango);

	if(mirango==0)
	{
		saludo_hijo=”Hola chaval”;
		MPI_Bcast(saludo_hijo,11,MPI_CHAR,0,MPI_COMM_WORLD);
		printf(“%.11s\n”,saludo_hijo);
	}
	else
	{
		MPI_Bcast(saludo_her,11,MPI_CHAR,0,MPI_COMM_WORLD);
		printf(“%.11s\n”,saludo_her);
	}
	printf(“Hola papa”\n”);
	MPI_Finalize();
	Return 0;
}
```

Diagrama de flujo de los programas

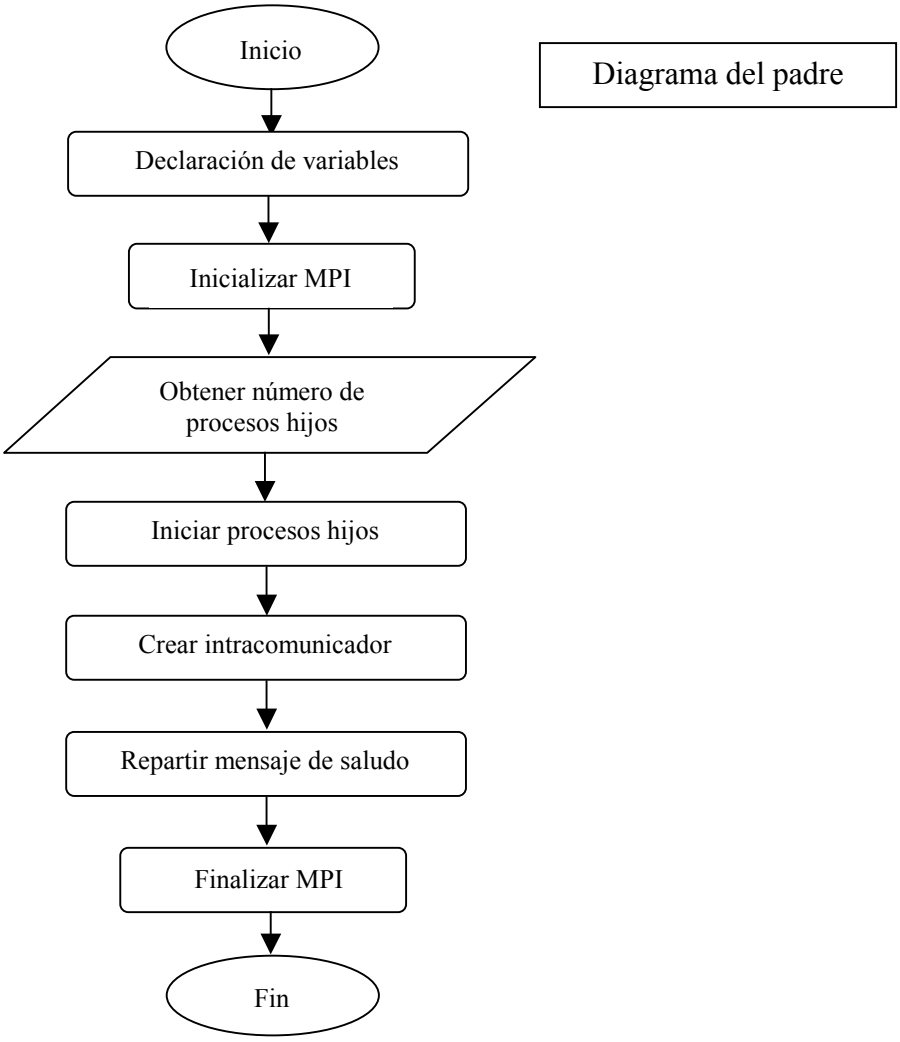
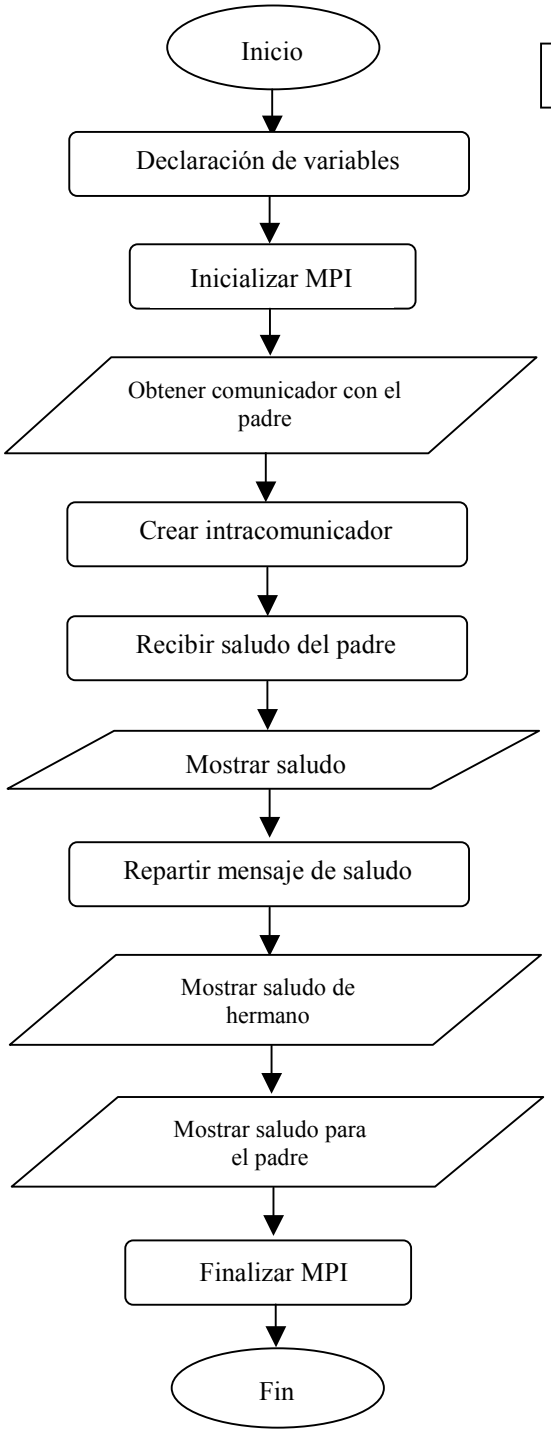


Diagrama del hijo



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.

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, la función MPI_Recv devuelve un parámetro del tipo MPI_Status que hasta el momento no hemos utilizado. Se trata de una estructura formada por tres elementos: MPI_SOURCE, MPI_TAG y MPI_ERROR. El primero proporciona la fuente del mensaje que se ha recibido; si se recibió con el valor MPI_ANY_SOURCE en la función MPI_Recv, puede resultar útil saber quién lo mandó. El segundo proporciona la etiqueta que traía el mensaje; si se recibió con el valor MPI_ANY_TAG en MPI_Recv, 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 para 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 para almacenar la mayor cantidad de datos posible.

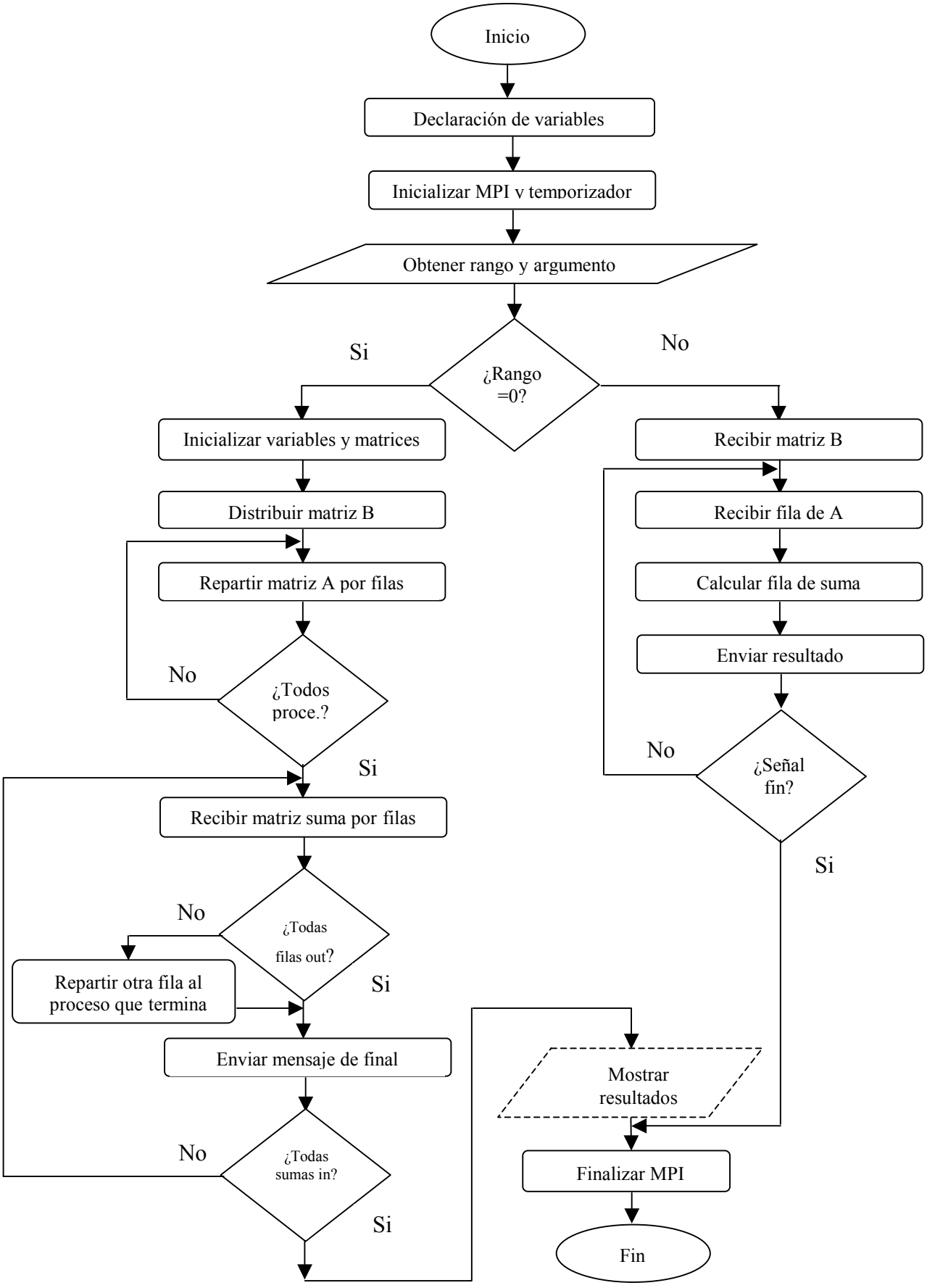
El proceso cero inicializará las matrices con valores aleatorios de tipo float y repartirá la carga de forma adecuada, según el criterio del programador. En una primera fase, se imprimirá el resultado en la salida estándar para comprobar que es correcto. Posteriormente se eliminará para permitir que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

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

#define FILAS 4000

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

{
      int mirango, tamano,indice;
      float *a[FILAS], *b[FILAS], *c[FILAS], *buffer, *val;
      int i,j,n,enviada,src,int_size,max_int,*asign;
      double sw_time, ew_time, tiempo;

      MPI_Status estado;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &mirango);
      MPI_Comm_size(MPI_COMM_WORLD, &tamano);

      if(mirango==0)
            sw_time=MPI_Wtime();
      n=atoi(argv[1]);

      int_size=sizeof (int) * 8;
      max_int=(exp(int_size*log(2)))/2 - 1;


      for(i=0;i<n;i++)
      {
      a[i] =(float *) malloc(n * sizeof (float));
      b[i] =(float *) malloc(n * sizeof (float));
      c[i] =(float *) malloc(n * sizeof (float));
      }
      buffer=(float *) malloc(n * sizeof (float));
      asign=(int *) malloc(n * sizeof (int));
      val=(float *)malloc(n * sizeof (float));

      srand(1);

      if(mirango==0)
```

```
{
      for(i=0;i<n;i++)
      {
            for(j=0;j<n;j++)
            {
             a[i][j]=rand()*2.0/max_int - 1.0;
            }
      }

      for(i=0;i<n;i++)
      {
            for(j=0;j<n;j++)
            {
             b[i][j]=rand()*2.0/max_int - 1.0;
            }
      }
}

for(i=0;i<n;i++)
      MPI_Bcast(b[i],n,MPI_FLOAT,0,MPI_COMM_WORLD);

if(mirango==0)
{
    i=0;
      enviada=0;
      while(i < tamano-1)
      {
            for (j=0;j<n;j++)
                  buffer[j]=a[i][j];
            MPI_Send(buffer, n, MPI_FLOAT, i+1, 10,
                     MPI_COMM_WORLD);
            asign[i+1]=i;
            i++;
            enviada++;
      }

      for(i=0;i<n;i++)
      {
            MPI_Recv(val, n, MPI_FLOAT, MPI_ANY_SOURCE, 20,
                     MPI_COMM_WORLD, &estado);
            src=estado.MPI_SOURCE;
            indice=asign[src];

            for(j=0; j<n; j++)
                  c[indice][j]=val[j];

            if (enviada<n)
            {
                  for (j=0;j<n;j++)
                        buffer[j]=a[enviada][j];
                  MPI_Send(buffer, n, MPI_FLOAT, src, 10,
                           MPI_COMM_WORLD);
                  asign[src]=enviada;
                  enviada++;
            }
            else MPI_Send(buffer, n, MPI_INT, src, 30,
                          MPI_COMM_WORLD);
      }
      ew_time=MPI_Wtime();
      tiempo=ew_time-sw_time;
      printf("\n\ndtime=%lf",tiempo);
```

```
	}
	else
	{
		do
		{
			MPI_Recv(buffer, n, MPI_FLOAT, 0, MPI_ANY_TAG,
				MPI_COMM_WORLD, &estado);
			if ((estado.MPI_TAG) == 30)
				break;
			else
			{
				j=0;
				while(j<n)
				{
					val[j]=0.0;
					for(i=0;i<n;i++)
						val[j]=val[j]+ buffer[i]*b[i][j];
					j++;

				}
			MPI_Send(val, n, MPI_FLOAT, 0, 20, MPI_COMM_WORLD);
			}
		} while ((estado.MPI_TAG) != 30);

	}

	if(mirango==0)
	{
		printf("\n");
		for(i=0;i<n;i++)
		{
			for(j=0;j<n;j++)
				printf("%f ",c[i][j]);
			printf("\n");
		}
	}

	MPI_Finalize();
	return 0;
}
```

Diagrama de flujo del programa



CUESTIONES

- Para multiplicar A x 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.

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

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. Son los siguientes:

- *Grado de paralelismo (DOP):* número de procesadores utilizados para ejecutar un programa en un tiempo determinado. Su representación temporal da lugar al perfil de paralelismo. No tiene porqué corresponder con el número de procesadores disponibles en el sistema. En las siguientes definiciones supondremos que se dispone de más procesadores que los necesarios para alcanzar el grado de paralelismo máximo de las aplicaciones.
- *Trabajo total realizado:* si Δ es la capacidad de trabajo de un solo procesador, medida en MIPS o MFLOPS y suponemos que todos los procesadores son iguales, es posible computar el trabajo total realizado a partir de área bajo el perfil de paralelismo como:

  $$W = \int_{t_1}^{t_2} DOP(t)dt$$

  Normalmente el perfil de paralelismo es un gráfica definida a tramos, por lo que se puede expresar el trabajo total realizado como:

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

Donde t_i es el tiempo durante el cual el grado de paralelismo es i. m es el máximo grado de paralelismo. Según esto, se cumple que:

$$\sum_{i=1}^{m} t_i = t_2 - t_1$$

- *Paralelismo medio:* no es otra cosa que la media aritmética del grado de paralelismo a lo largo del tiempo de trabajo. Se expresa como:

$$A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} DOP(t) \cdot dt = \frac{\sum_{i=1}^{m} i \cdot t_i}{\sum_{i=1}^{m} t_i}$$

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

- *Incremento máximo 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, $t_i(k) = \frac{W_i}{k \cdot \Delta}$; para infinitos procesadores, $t_i(\infty) = \frac{W_i}{i \cdot \Delta}$. A partir de aquí se puede definir el tiempo de respuesta como: $T(1) = \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m} \frac{W_i}{\Delta}$ y $T(\infty) = \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m} \frac{W_i}{i \cdot \Delta}$. 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 alcanzable se obtendrá del cociente entre éste y el rendimiento que proporciona un solo procesador. En términos de tiempos de respuesta se expresa como:

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

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.

Otra forma de evaluar el incremento de rendimiento se basa en considerar que un trabajo, 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. R_i representaría la velocidad colectiva de todos ellos medida en MIPS o MFLOPS; R_1 sería la velocidad de uno solo y T=1/R el tiempo de ejecución. Supongamos que el trabajo se realiza en "n" modos diferentes con carga de trabajo diferente para cada modo, lo cual se refleja en un peso relativo f_i que se le asigna a cada uno. En estas condiciones se define nuevamente el incremento de rendimiento como:

$$S = \frac{T_1}{T^*} = \frac{1/R_1}{\sum_{i=1}^{n} f_i/R_i}$$

Si nos vamos a una situación ideal en la que no existen retardos por comunicaciones o escasez de recursos tendremos que si R_1=1, R_i=i, de forma que:

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

Quedando en este caso una expresión análoga a la vista anteriormente.

A partir del caso anterior se enuncia la ley de Amdahl. En este caso se parte de que $R_i = i$ y se supone que $W_1 = \alpha$ y $W_n = 1-\alpha$ lo que implica que el trabajo se va a realizar parte en modo secuencial y el resto empleando la potencia máxima del sistema, de manera que en ningún momento se emplea solamente una parte del sistema. En estas condiciones:

$$S_n = \frac{1}{\frac{\alpha}{1} + \frac{1-\alpha}{n}} = \frac{n}{1+(n-1)\alpha}$$

Esto implica que:

$$n \to \infty \Rightarrow S \to 1/\alpha$$

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

- *Eficiencia:* determina el grado de aprovechamiento del sistema disponible:

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

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

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

- *Utilización:* refleja el grado de utilización , como su propio nombre indica, del sistema completo:

$$U = R \cdot E = \frac{O(n)}{n \cdot T(n)}$$

- *Calidad del sistema:*

$$Q = \frac{S \cdot E}{R} = \frac{T^3(1)}{n \cdot T^2(n) \cdot O(n)}$$

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 y 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. Podemos partir por ejemplo de matrices 1000x1000. Esto supone un volumen de cálculo de 10^9 multiplicaciones. Es de esperar que para ordenadores Pentium IV a 1'8 GHz el

trabajo en paralelo de varias máquinas no suponga un incremento significativo respecto al empleo de una sola.

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, dos, tres,.. máquinas. 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

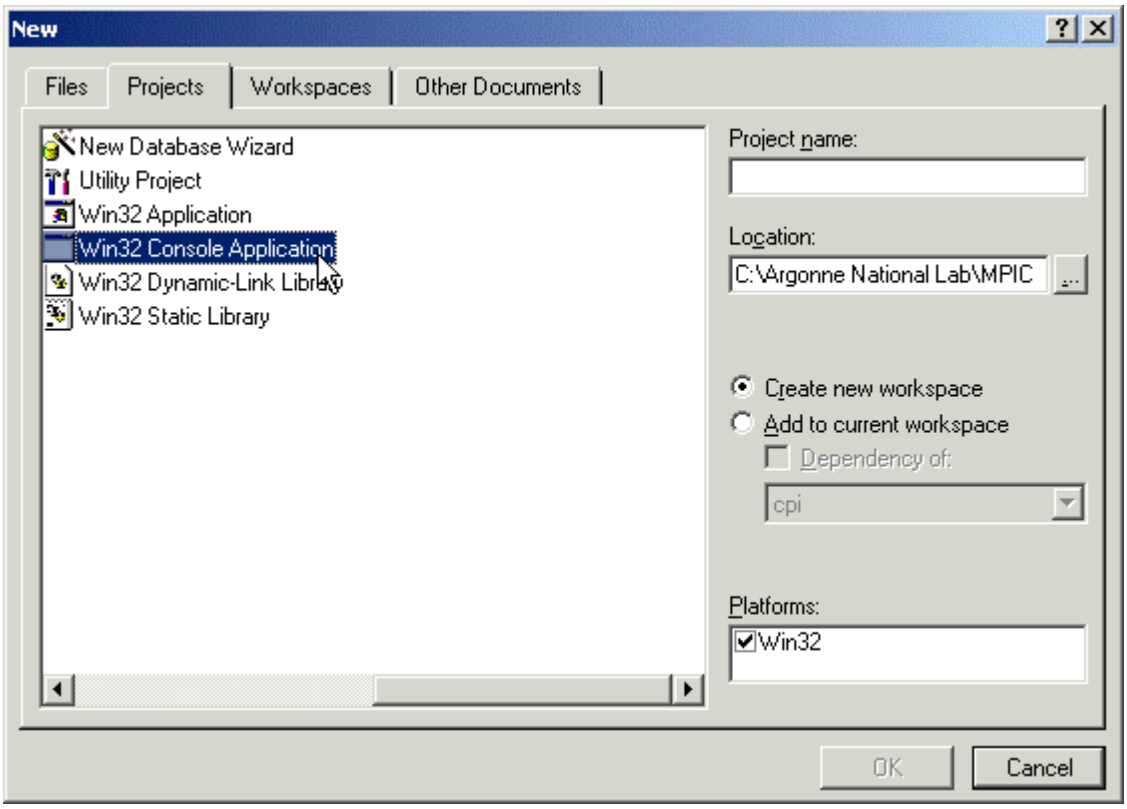
- Evaluar el grado de acercamiento de la aplicación desarrollada a los máximos posibles.
- Evaluar para este caso concreto: eficiencia, utilización, redundancia y calidad de forma aproximada.
- ¿Está muy lejos el rendimiento obtenido del máximo teórico que se puede alcanzar?
- Estimar las causas de la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

APÉNDICE A

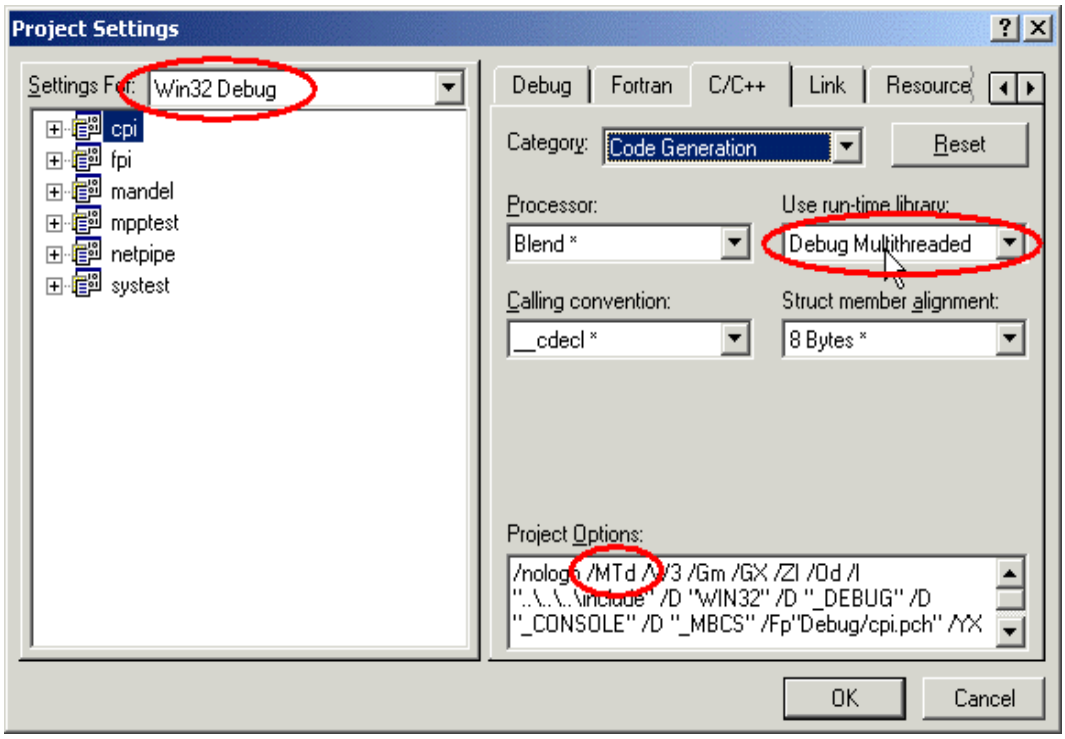
CONFIGURACIÓN DE PROYECTO EN VISUAL C++

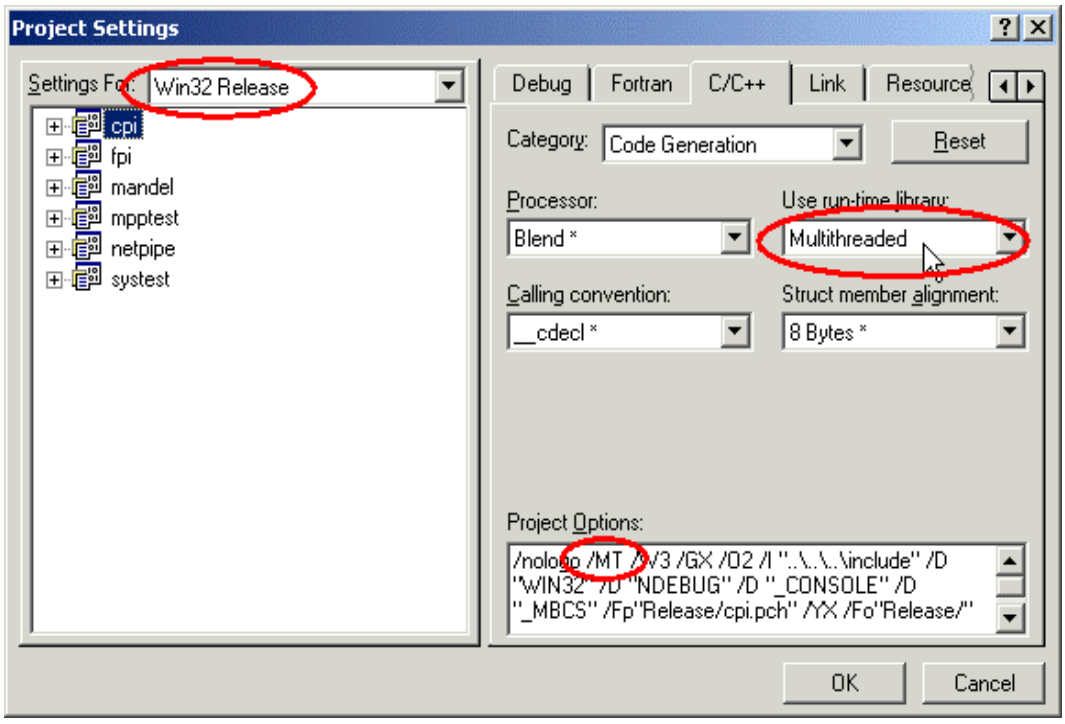
A continuación se describe de forma gráfica los pasos necesarios para configurar un proyecto de Visual C++ para que se puedan generar aplicaciones escritas en C que hagan uso de las librerías de MPI para Windows. Se trata simplemente de seguir los pasos que se detallan:

1. Abrir MS Developer Studio - Visual C++
2. Crear un nuevo proyecto con el nombre y ruta que se desee. Se creará como aplicación de consola vacía.

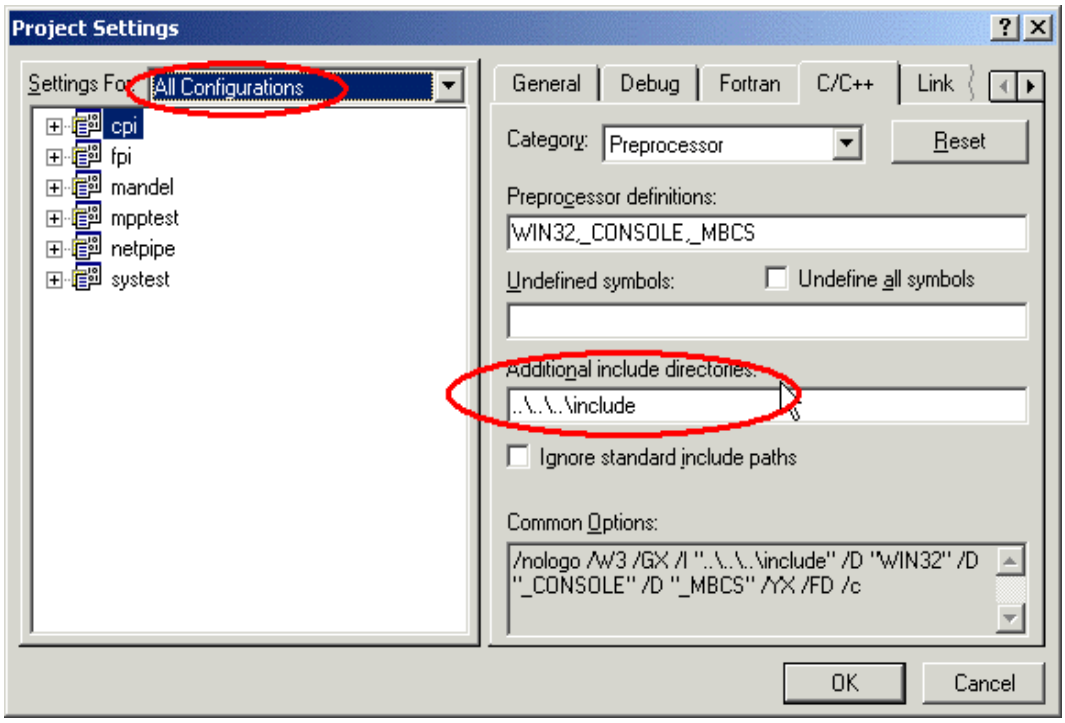


3. Finalizar el asistente de nuevo proyecto.
4. Ajustar las propiedades del proyecto mediante Project->Settings o pulsar Alt F7 según se describe en los siguientes pasos.
5. Cambiar las propiedades para usar librerías multitarea (multithreaded). Cambiar las propiedades para los objetivos Debug y Release.

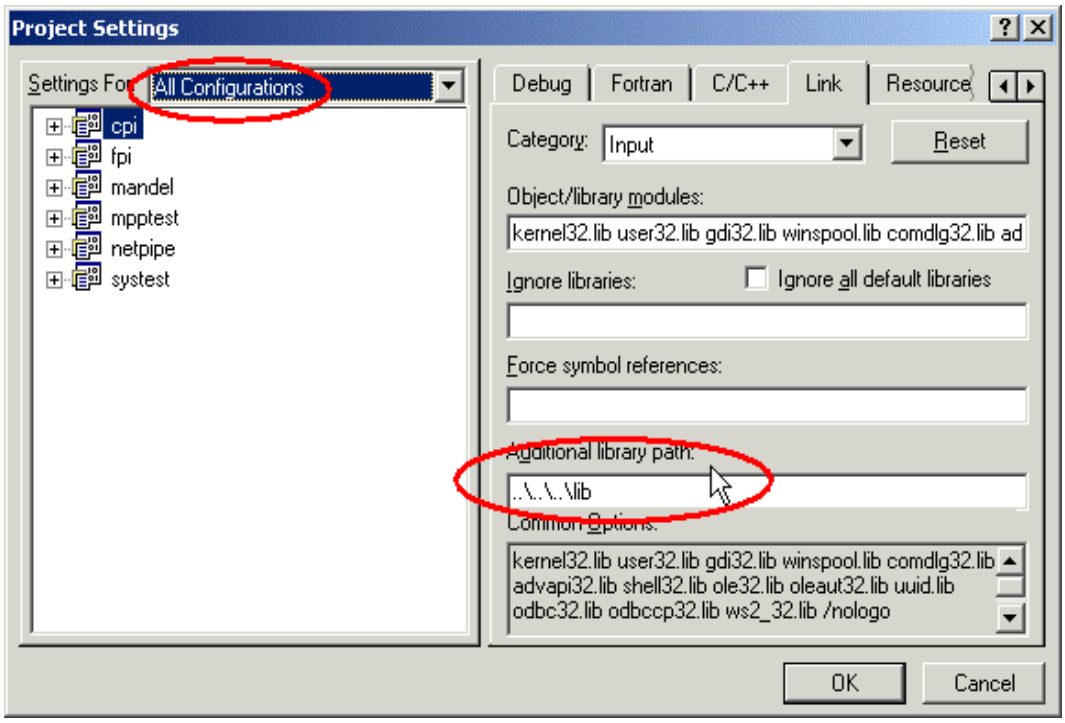




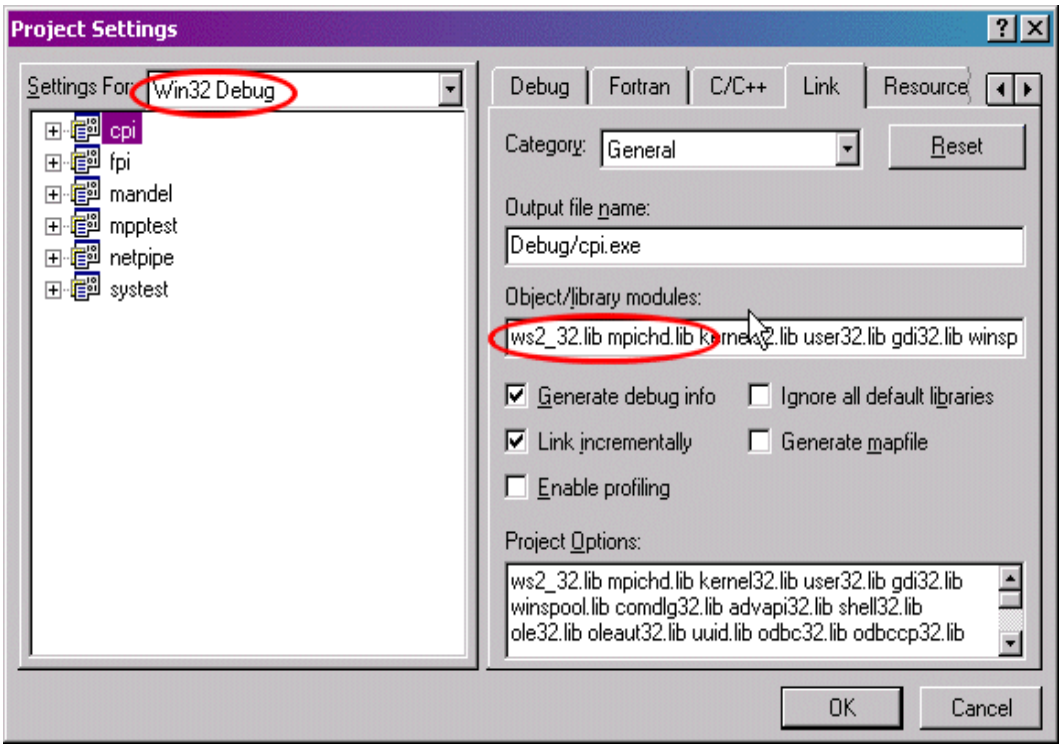
6. Establecer el directorio “include” para todas las configuraciones objetivo: Debería ser Argonne National Lab\MPICH.NT.1.2.1\SDK\include

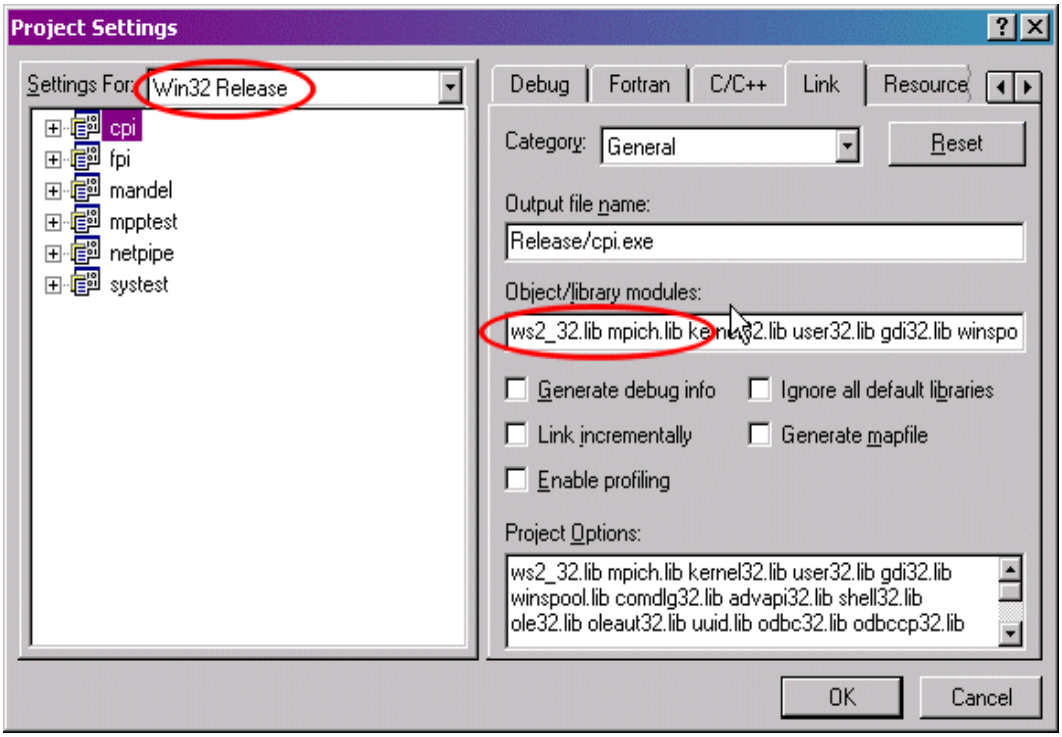


7. Establecer el directorio “lib” para todad las configuraciones objetivo.: Debería ser Argonne National Lab\MPICH.NT.1.2.1\SDK\lib



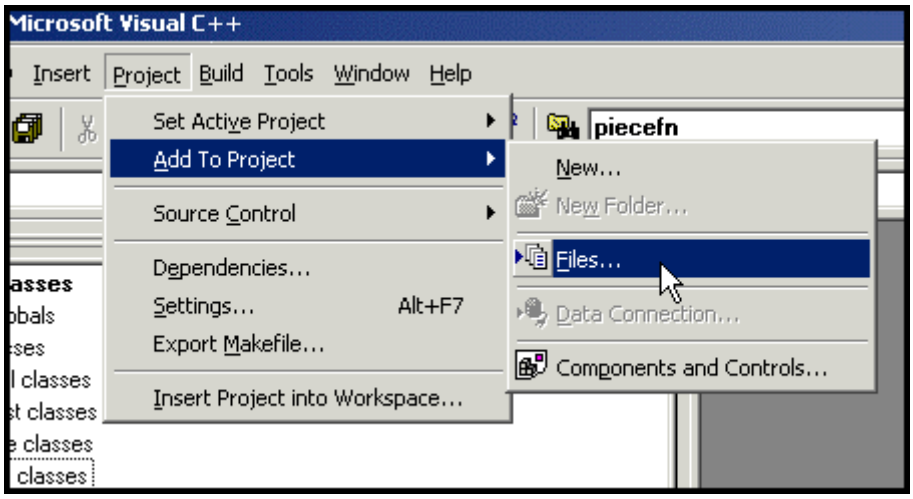
8. Añadir la biblioteca ws2_32.lib a todas las configuraciones (Esta es la librería Microsoft Winsock2. Está en el directorio por defecto para las librerías). Añadir mpich.lib al objetivo release y mpichd.lib al objetivo debug.





9. Cerrar el cuadro de diálogo de las propiedades del proyecto.

10. Añadir los ficheros fuente al Proyecto (opción Files) o crearlos (opción New).



11. Construir.

Las fuentes bibliográficas empleadas para la elaboración de este proyecto se reducen principalmente a tres aspectos: un libro de referencia sobre los principales conceptos relacionado con las arquitecturas avanzadas de computadores, la consulta de innumerables páginas web con información de todo tipo y la documentación descargada de internet en forma de diferentes tipos de archivos: .pdf, .doc, .ps y .ppt principalmente.

El libro empleado como referencia ha sido:

ADVANCED COMPUTER ARCHITECTURE

Kai Hwang. Ed. Mc Graw Hill

Se trata de un clásico en la materia. Contiene capítulos muy completos y exhaustivos sobre todos los conceptos relacionados con las arquitecturas avanzadas: superescalares, vectoriales, solapadas, paralelas... No obstante, su estructura dispersa mucho los contenidos, por lo que siendo y muy buen libro de consulta, no resulta apropiado como material introductorio; su lectura sin un conocimiento previo de la materia puede resultar frustrante.

En cuanto a las páginas web consultadas, sería prácticamente imposible relacionarlas todas. En la lista que se aporta a continuación, aparece aquellas que por su abundancia de información, fiabilidad, profundidad y claridad se ha decidido conservar para posteriores consultas. En algunas de ellas se proporcionan completas publicaciones en formato web cuya lectura se recomienda para aquellas personas interesadas en introducirse en los temas relacionados con este proyecto. El principal problema de las direcciones de Internet como fuentes bibliográficas es su carácter efímero. Es posible que algunos intentos de consulta basados en la relación que aquí se incluye resulten en fracaso por desaparición del sitio correspondiente. Esto es habitual sobre todo en aquellos que son soportados por particulares o incluso universidades. No obstante, se puede intentar localizarlos en cachés de buscadores o en las páginas principales de los organismos que los soportan. Algunos de los sitios más interesantes consultados ya habían desaparecido en el momento de recopilar esta lista, de manera que no se han incluido ya en ella. Otros sobre los que no se tiene seguridad de su desaparición definitiva se han incluido en la lista sin comentar con la esperanza de que la ausencia de respuesta se deba a caídas temporales de los correspondientes servidores.

No aparece en la lista la información consultada sobre aspectos puntuales que han supuesto los principales obstáculos en el desarrollo del proyecto. Los foros y listas de preguntas frecuentes han permitido solucionar muchos de los problemas sobre todo de instalación y configuración de software. Ha resultado gratificante ver cómo otras personas se estaban enfrentando en el mismo momento a muchos de los problemas que han ido apareciendo en el desarrollo del proyecto. Sobre algunos de ellos se han encontrado soluciones; sobre otros ha sido necesario investigar y encontrar la explicación a preguntas que otros usuarios se estaban haciendo en ese mismo instante en diferentes partes del mundo.

Veamos cuáles son las principales fuentes de información consultadas:

http://aeroguy.snu.ac.kr/links/parallel_main.htm

http://araxa.lecom.dcc.ufmg.br/cursos/dcc007/
Apuntes sobre tecnología de computadores de una universidad brasileña. Está en portugués.

http://archive.ncsa.uiuc.edu/Alliances/Exemplar/Training/NCSAMaterials/IntroParallel_II/
Libro web sobre computación en paralelo. Universidad de Ilinois

http://archive.ncsa.uiuc.edu/SCD/Hardware/CommonDoc/MessPass/
Introducción al modelo de paso de mensajes. Incluye PVM, MPI y sus ámbitos de aplicación.

http://cag-www.lcs.mit.edu/6.004/Lectures/
Presentaciones sobre diversos aspectos de la arquitectura de computadores. Instituto Tecnológico de Massachussets

http://ciips.ee.uwa.edu.au/~morris/CA406/
Apuntes sobre la arquitectura de los procesadores modernos.

http://citeseer.nj.nec.com/directory.html
Página de NEC con información de multitud de aspectos relacionados con la computación, tanto a nivel hardware como software.

http://csep1.phy.ornl.gov/ca/ca.html
Libro web sobre arquitectura de computadores, tanto básica como avanzada.

http://dewww.epfl.ch/~debes/inter.html

http://nacphy.physics.orst.edu/PVM/pvm.html

http://usgibm.nersc.gov/mpi/d3d80mst02.html
Página de IBM sobre MPI. Contiene una breve descripción de todas las funciones incluidas en el estándar MPI-2.

http://winpar.iit.uni-miskolc.hu/onldoc/wpvm/

http://www.cray.com
Página web del fabricante de supercomputadores Cray. Incluye características completas de algunas máquinas.

http://www.cs.arizona.edu/classes/cs620/sp00/Processors/

http://www.cs.bris.ac.uk/Teaching/Resources/COMS11200/pages/tour6/sld001.htm
Presentación sobre técnicas de incremento de rendimiento en computadores.

http://www.cs.iastate.edu/~prabhu/Tutorial/title.html
Tutorial sobre jerarquía de memoria y arquitecturas solapadas.

http://www.cs.umn.edu/Research/Agassiz/agassiz_pubs.html
Enlaces a artículos sobre arquitecturas multihilo.

http://www.cse.uiuc.edu/cse302/
Página de la universidad de Ilinois con ejemplos de programas MPI.

http://www.csm.ornl.gov/pvm/pvm_home.html
Página oficial de PVM.

http://www.csse.monash.edu.au/~davida/teaching/cse3304/Web/Chapter7/
Presentación sobre multitud de aspectos relacionados con las arquitecturas superescalares.

http://www.epcc.ed.ac.uk/epcc-tec/documents/techwatch-mpi2/MPI2-1.html

http://www.hispacluster.org
Es un portal en castellano sobre clusters de computadores. Tiene mucha información y enlaces

http://www.ieeetfcc.org/
Página oficial del IEEE sobre clusters.

http://www.informatik.uni-stuttgart.de/ipvr/bv/cppvm/online-doc/
Página sobre programación en C++ con PVM.

http://www.jics.utk.edu/parallel.html
Contiene información y enlaces sobre computación paralela y avanzada.

http://www.links2go.com/topic/Parallel_Computing

http://www.linuxdoc.org/HOWTO/Parallel-Processing-HOWTO.html
Apuntes sobre computación en paralelo en LINUX.

http://www.markus-fischer.de/
Página personal de un experto en computación paralela. Incluye información y consejos sobre PVM y MPI. A pesar de ser una página personal, su contenido es fiable.

http://www.microway.com/papers/appnotes/Microway_Working_Note_Index.htm
Notas de aplicación sobre multitud de aspectos relacionados con MPI y clusters.

http://www.netlib.org/utk/papers/latbw/commperf.html
Estudio comparativo sobre el empleo de paso de mensajes en varios computadores. Viene firmada por Jack Dongarra, uno de los grandes expertos en la materia.

http://www.npac.syr.edu/copywrite/pcw/
Libro web sobre computación paralela y algunos enlaces adicionales.

http://www.pdc.kth.se/training/Talks/SMP/maui98/message_passing/message_passing.html
Documento introductorio sobre los conceptos básicos de la computación paralela.

http://www.personal.psu.edu/faculty/l/n/lnl/424/
Curso sobre métodos numéricos en computadores paralelos.

http://www.phys.uu.nl/~steen/web00/overview00.html
Documento comparativo de características de supercomputadores actuales. Mantiene actualizada la lista. Firmado por Jack Dongarra.

http://www.scripps.edu/~olivier/beowulf.html
Información y enlaces sobre clusters Beowulf.

http://www.stanford.edu/class/ee382/vector/ppframe.htm
Presentación sobre arquitecturas SIMD y vectoriales.

http://www.wikipedia.com/wiki/Vector_processor
Documento sobre procesadores vectoriales y conceptos relacionados.

http://www-unix.mcs.anl.gov/dbpp/text/book.html
Libro web sobre diseño de programas en paralelo.

http://www-unix.mcs.anl.gov/mpi/
Página oficial de MPI.

Sobre los documentos descargados de internet, se va a proporcionar una breve descripción de los más interesantes. En el CD-ROM adjunto se proporcionan los ficheros recogidos. Algunos se han cambiado de nombre para facilitar una referencia rápida a sus contenidos. El nombre proporcionado en esta relación coincide con el que tienen en el CD-ROM.

arvanced_scientific_computing.pdf

Contiene información sobre procedimientos de computación paralela aplicados a diversas materias científicas. Finaliza con una referencia a los modelos de paso de mensajes.

building_beowulf_note.pdf

Referencia de instalación de un cluster en Linux. Este tipo de documentos resultan muy genéricos, por lo que para una instalación en un sistema análogo al de este proyecto resulta más apropiado el anexo de instalación que se adjunta con él. En caso contrario este documento puede constituir una referencia muy interesante.

CC-Edu.pdf

Es un artículo muy interesante sobre la enseñanza de computación en cluster. Justo lo que pretende el proyecto.

EuroPVM98.pdf

Describe la configuración de clusters PVM en Windows NT. Esta opción, considerada como poco viable en este proyecto, es comentada en este artículo haciendo referencia a algunos de los problemas que efectivamente se han encontrado en las pruebas realizadas en Windows.

final-paper.pdf

Es un completísimo documento sobre computación en cluster. Abarca aspectos de hardware, software, comunicaciones, sistemas operativos, administración de sistemas. Realiza un estudio a fondo y con ánimo crítico.

ft-mpi.pdf

Documento descriptivo de una implementación prototipo de MPI tolerante a fallos desarrollada por la Universidad de Tenesse.

mpi.guide.pdf

Proporciona una descripción de las diferentes funcionalidades de MPI, explicando el empleo de muchas de las funciones disponibles.

mpiandpvm.pdf

Es una referencia obligada a la hora de comparar las prestaciones de PVM y MPI.

mpi-com.pdf

Documento descriptivo de una implementación de MPI orientada a objeto.

mpichman.pdf

Contiene guía de instalación y manual de usuario de la implementación MPICH de MPI.

mpi-objeto.pdf

Librería de clases para MPI. Se trata de otra implementación de MPI orientada a objeto.

mpi-quickref.pdf

Referencia rápida de funciones MPI. No contiene descripción.

pe_mpi_sub_ref_v3r10.pdf

Referencia completa de funciones de MPI. Se basa en la implementación de IBM para AIX.

pvm-mpi.pdf

Introducción esquemática a los sistemas de paso de mensajes: PVM y MPI.

PVMvsMPI.pdf

Artículo convertido en clásico en el que se realiza un estudio comparativo de PVM y MPI.

trace.pdf

Documento sobre monitorización en PVM.

Intro_MPI.ppt

Introducción al funcionamiento de MPI. Incluye funciones de MPI-1

mpich.nt.ppt

Incluye aspectos técnicos de funcionamiento de MPI para Windows NT.

Parallel_IO_MPI_2.ppt

Explica el funcionamiento de la entrada-salida paralela en MPI-2.

WindowsClusters-0.8.doc

Estudio comparativo de diferentes tipos de clusters tanto en Windows como en Linux.

ANEXO
ESTUDIO ECONÓMICO

No se va a realizar un presupuesto en toda regla del coste del sistema planteado en el proyecto debido a que buena parte de los apartados habituales en cualquier presupuesto van a ser realizados por personal de la propia institución universitaria incluido el autor del mismo. Por otro lado, la infraestructura e instalaciones complementarias: eléctrica, de red, seguridad, etc, se encuentran ya disponibles.

En estas circunstancias el estudio económico se simplifica mucho. Se trata únicamente de plantear el coste de los equipos informáticos de nueva adquisición. Dado que los grupos de prácticas están compuestos por un máximo de 24 personas, se podría considerar como correcto un sistema formado por 12 puestos de trabajo. Dos personas por ordenador en el peor de los caso no parece excesivo. No obstante, las limitaciones presupuestarias han obligado a reducir este número a 6, circunstancia que penaliza considerablemente el desarrollo de las prácticas, lo que obliga a plantearse soluciones alternativas como la creación de grupos extra de prácticas u otras posibilidades.

En cualquier caso, el gasto realizado se reduce a los mencionados ordenadores, el software necesario y se plantea también la adquisición de un "switch" de 10/100 Mbps. Todo ello se refleja en la siguiente relación:

Concepto	Cantidad	Precio unitario	Precio total
Ordenador de sobremesa[1]	6	926 €	5566 €
Sistema operativo Windows 2000 Profesional[2]	6	169'24 €	1015'44 €
Microsoft visual estudio 2002 profesional[2]	6	1197'45 €	7184'7 €
Switch 10/100 Mbps OvisLink	1	445'26 €	445'26 €
		Total	14211'4 €
		IVA 16%	2273'82 €

[1] Características principales: Intel Pentium 4, 1'8 GHz, 256 Mb de memoria RAM DDR, disco duro de 40 Gb, CR-ROM x52, monitor 17".

[2] Precios de licencias para profesionales, sin aplicar descuentos para docencia.

ANEXO
INSTALACIÓN DE MPI

El proceso de instalación de MPI difiere según se quiera trabajar en Windows o en LINUX, los dos sistemas considerados en este proyecto.

INSTALACIÓN EN WINDOWS

Para instalar MPI bajo Windows NT, 2000 o XP se requiere una cuenta de administrador. Este nivel de privilegio es necesario solamente para la instalación, no así para el trabajo normal. Para otros sistemas operativos de Microsoft (Windows 3.x, 9x) no existe una versión de MPI disponible.

El proceso de instalación es muy sencillo ya que el software dispone de su propio programa de instalación como cualquier aplicación Windows. Se puede obtener este paquete libremente en la dirección: http://www-unix.mcs.anl.gov/mpi/

De esta forma obtendremos la implementación MPICH de MPI, una de las más extendidas y la única disponible para Windows de forma gratuita.

Una vez instalado el software y supuesto que los ordenadores que forman parte del cluster se comunican correctamente, se puede comenzar a trabajar en el desarrollo y ejecución de aplicaciones paralelas. Es importante aclarar que, siguiendo con la filosofía de “cluster” es necesario disponer de un usuario idéntico en todas las máquinas en caso de no tener un dominio NT configurado. Es recomendable que los programas que se van a ejecutar se encuentren instalados en la misma ruta en todas las máquinas.

El programa de instalación dejará una serie de utilidades en la carpeta correspondiente, de las cuales la principal es MPIRUN. Se trata de una interfaz gráfica que permite al usuario lanzar las aplicaciones creadas. Dispone de algunas opciones de configuración que conviene mencionar:

- Hosts: deberemos especificar la lista de máquinas entre las que queremos que se repartan los procesos que se van a lanzar. No se buscan automáticamente, de manera que habrá que añadir las máquinas remotas una a una por su nombre. Para ello deben poder reconocerse por nombre en la red local.

- Advanced options: solamente una de estas opciones es imprescindible marcar. Es la que exige la introducción de password para poder lanzar las aplicaciones. Esto permite la identificación en cada una de las máquinas del “cluster”
- Application: nos pide la ruta completa de la aplicación que queremos lanzar. Esta ruta deberá ser idéntica en todas las máquinas.
- Number of processes: se emplea para especificar cuántos procesos de quieren lanzar. De forma trnasparente se asignarán procesos a máquinas configuradas.
- Run: lanza la aplicación una vez establecidas las opciones de configuración.

La creación de aplicaciones MPI para Windows es también bastante sencilla. Se requiere un compilador C (Visual C++), siendo posible la generación de aplicaciones escritas en C++. En la introducción del manual de prácticas incluido en este proyecto se explica cómo configurar los proyectos creados en Visual C++ para la correcta construcción de aplicaciones MPI.

INSTALACIÓN EN LINUX

El proceso de instalación de MPI en LINUX tampoco es excesivamente complicado, aunque sí requiere de más acciones por parte del usuario, que al igual que en Windows necesitará cuenta de superusuario (root) para la instalación.

En primer lugar, se debe señalar que existen varias implementaciones de MPI para LINUX. A pesar de que MPI dispone de un estándar estructurado en dos partes: MPI y MPI-2, no todas las implementaciones son iguales, ya que no tienen porqué implementar todas las funciones señaladas en el estándar. En este proyecto se ha ensayado la instalación y configuración de dos implementaciones: MPICH y LAM-MPI.

Otra cuestión importante es que la aplicación final que proporcionan ambas implementaciones (MPIRUN) es únicamente una aplicación de consola, sin interfaz gráfica. Existe una interfaz gráfica adicional a nivel de monitorización y depuración (XMPI) que se ha ensayado también en el proyecto.

Antes de iniciar el proceso de instalación conviene configurar algunos aspectos importantes para el funcionamiento del software. Al igual que ocurre en Windows, la comunicación entre las máquinas se basa por defecto en el empleo del protocolo RSH (shell remoto) para la ejecución de comandos en otras máquinas. Este protocolo forma

parte de TCP/IP, por lo que se encuentra instalado en LINUX. No obstante, conviene asegurarse de que el servicio RSH, tanto cliente como servidor se inician correctamente. Existe la posibilidad de emplear otros protocolos alternativos a RSH como es SSH; nos vamos a centrar en el primero. Además de iniciar el servicio debemos especificar a qué máquinas se les va a permitir el acceso a cada máquina local. Para ello existe un fichero denominado ".rhosts" situado en el directorio "home". En este fichero se añadirá una línea por cada máquina desde la que se quiera permitir el acceso. La sintaxis será:

máquina usuario

Así, si queremos que el usuario "arturo" se pueda conectar desde la máquina "cuatro" escribiremos:

cuatro arturo

Puede resultar necesario configurar asimismo el fichero etc/host.equiv incluyendo en él una lista con las máquinas remotas desde las cuales el acceso a cada máquina local va a estar permitido. Incluso resulta conveniente comprobar la configuración de los ficheros /etc/host.allow y /etc/host.deny para evitar que alguna de las máquinas pueda estar excluida.

Si el servicio de shell remoto no funciona correctamente, de nada servirá que se continúe con la instalación, de manera que será conveniente verificar su funcionamiento. Para ello se puede realizar un acceso remoto entre las diferentes máquina configuradas empleando el comando "rsh". Su sintaxis es la siguiente:

rsh –l login máquina

Por supuesto, tiene más opciones, pero con esta prueba nos bastará para probar su funcionamiento. En este caso, si el usuario "arturo" quiere comprobar que desde la máquina "cuatro" se le permite el acceso por ejemplo a la máquina "dos" ejecutará:

rsh –l arturo dos

Si el servicio funciona correctamente obtendrá un mensaje de saludo y un "prompt" en su directorio "home" de la máquina "dos". Podrá a partir de este momento trabajar en la máquina remota como en la suya propia, aunque en este momento esto no sirve para nada, por lo que una vez verificado el funcionamiento, se saldrá con el comando "exit" retornando con ello a la máquina local.

Normalmente el servicio de shell remoto deberá funcionar sin problemas. En caso de que esto no ocurra se podrá comprobar lo siguiente:

1. El servicio de shell remoto tanto cliente como servidor está instalado en todas las máquinas. En caso de no estarlo, se deberá instalar.
2. Se ha configurado el arranque de los servicios. Para verificarlo se deberá acudir al fichero “inetd.conf” situado en el directorio “etc”. En este fichero se especifican los servicios de red que se arrancarán al iniciar la máquina.
3. Los servicios se han iniciado. No es suficiente verificar que se ha configurado el arranque de los servicios; puede ser que éste no haya tenido éxito. Habrá que comprobar el estado de los servicios de red, con especial atención al shell remoto. No debería ser necesario, pero se puede intentar iniciar de forma manual para comprobar que el problema se soluciona.

Si a pesar de que estas comprobaciones son correctas, el shell remoto sigue sin funcionar habrá que culpar al estado y configuración de la red local, tanto a nivel de hardware como de software, prestando especial atención al servicio de resolución de nombres. Es recomendable comprobar la existencia de comunicación con las máquinas remotas, por ejemplo mediante el comando “ping”. Si se recibe respuesta, cabe achacar el fallo a la configuración de la resolución de nombres, ya sea vía DNS o mediante el fichero /etc/hosts; de lo contrario habrá que pensar en la configuración de la red e incluso en el hardware.

Una vez que se tiene el shell remoto en funcionamiento se puede continuar con el proceso. Para ello se debe disponer de una implementación de MPI para LINUX. Comenzaremos por instalar completamente la implementación de “Argonne National Lab” (MPICH) para UNIX. En estos momento está disponible la versión 1.2.2.3 en la dirección: www.mcs.anl.gov/mpi/mpich/download.html . Se obtiene como fichero comprimido: mpich-1.2.2.3.tar.gz. A partir de este fichero se va a comenzar el proceso de instalación para lo cual se necesitarán permisos de superusuario. Seguiremos los siguientes pasos:

1. Se coloca el fichero en /usr/local/ y se descomprime el fichero: tar –xzvf mpich-1.2.2.3.tar.gz
2. Se accede al nuevo directorio creado: cd mpich-1.2.2.3

3. Se establecen las variables de entorno necesarias para configurar la instalación. Existen numerosas opciones, aunque suele ser suficiente con la siguiente: ./configure –prefix=/usr/local/mpich-mpi . Esto hará que el software se instale en esta ruta. Puede haber problemas si se intenta instalar en la misma ruta de los ficheros fuente.
4. Se compila el software: make
5. Se realiza la instalación: make install
6. Aunque el proceso de instalación ha debido finalizar correctamente, aún es necesario realizar algunas tareas de configuración para poder empezar a trabajar. En primer lugar, debemos incluir en el "path" la ruta de los ficheros binarios, tanto de mpi como de aplicaciones de usuario. Para ello en el fichero /home/.bashrc o en el correspondiente a la shell que se esté empleando añadiremos:

   PATH=$PATH:/usr/local/mpich-mpi/bin

   PATH=$PATH:/usr/local/mpich-mpi/sbin

   export PATH
7. Es necesario también especificar qué máquinas formarán parte del cluster. Esto se configura en el fichero:

   /usr/local/mpich-1.2.2.3/util/machines/machines.LINUX

   En él colocaremos el nombre de cada máquina en una línea. En caso de que alguna máquina disponga de más de un procesador, se pondrá el número de ello separado mediante dos puntos del nombre de la máquina. Si en nuestro cluster tenemos el ordenador "dos" y el "cuatro" y éste último dispone de dos procesadores podremos tener:

   dos

   cuatro:2

   La primera vez que abramos este fichero veremos que se encuentra el nombre de la máquina local escrito varias veces. Esto se puede modificar, aunque se puede mantener la repetición del nombre de alguna máquina. Esto guiará a MPI a la hora de asignar máquinas a procesos.

Siguiendo los pasos anteriormente descritos, se estará en condiciones de empezar a trabajar con MPI. Partiendo de un fichero fuente en C, ya sea propio o tomado de los ejemplos que se adjuntan con el software, se creará un ejecutable y se probará a ejecutarlo con diferente número de procesos. Esto requiere varios pasos:

1. Compilación: mpicc mi_programa.c –o mi_programa
2. Ejecución: convendrá situar el ejecutable en ../mpich-mpi/bin/. Se escribirá: mpirun mi_programa
3. Pruebas: con mpirun se puede especificar el número de procesos que se quieren lanzar para forzar así la inclusión de otras máquinas: mpirun –np número mi_programa

El principal problema de la implementación MPICH es que no abarca el estándar MPI-2 completo, dejando fuera de su ámbito la creación dinámica de procesos, es decir, el lanzamiento de procesos “hijos” desde un determinado proceso. Este es un aspecto fundamental que tradicionalmente ha apartado a MPI de muchas de las aplicaciones que se estaban desarrollando con PVM. Más aún, se podría decir que ésta es la diferencia conceptual entre la máquina virtual y el sistema de paso de mensajes.

LAM MPI desarrollado por la Universidad de Notre Damme y la Universidad de Ohio implementa el estándar completo, por lo que se va a documentar su instalación y empleo.

En primer lugar, el software se puede obtener en: www.lam-mpi.org Se trata de un centro de desarrollo muy dinámico hasta el momento, por lo que es interesante echar un vistazo a las muevas versiones disponibles así como a las recomendaciones en cuanto a la prueba de versiones beta etc.

Una vez descargado el paquete comprimido se procede a su instalación siguiendo los paso habituales que no obstante se van a detallar:

1. Habrá que descomprimir el paquete (lam-6.5.6.tar.gz), en este caso se trata de la versión 6.5.6 a un directorio. Como en el caso de MPICH se ha instalado en /usr/local para que todos los usuarios tengan acceso a él.
2. A continuación se proporcionan las opciones de configuración, que pueden ser muchas, pero habitualmente basta con proporcionar la ruta para su instalación. En este caso, se ha comprobado que resulta problemático instalar el software en

la misma ruta en la que se encuentran los fuentes, por lo que es recomendable proporcionar una ruta alternativa. No es necesario que exista, en tal caso el script de instalación la crearía. Por ejemplo, si el software se descomprimió a la ruta /usr/local/lam-6.5.6, se podría configurar la instalación de la siguiente manera:

./configure --prefix=/usr/local/lam-mpi. Si se pretende instalar XMPI posteriormente, será necesario incluir también la opción –with-trillium.

3. Se compila el software: make
4. Se realiza la instalación: make install
5. Una vez instalado el software son necesarias algunas operaciones adicionales. Al igual que en MPICH será necesario configurar la ruta de los ficheros binarios para que puedan ser invocados desde cualquier directorio. Para ello, en home/.bashrc se escribirá:

   PATH=$PATH:/usr/local/lam-mpi/bin

   export PATH

   Si previamente se había instalado MPICH se deberá eliminar del path para que no haya conflictos a la hora de buscar ficheros que tienen el mismo nombre en las dos implementaciones como “mpirun”.
6. Existen ficheros de configuración de las máquinas que forman parte del cluster, aunque se recomienda no modificar su contenido por defecto, de manera que la forma práctica de incluir los nodos remotos consiste, bien en no especificar nada con lo que se incluirán todas las máquinas que se han encontrado; o bien en emplear un fichero con la lista de máquinas que se van a incluir en el momento del arranque.

Acabamos de hablar del arranque. Este es un proceso que se hace en PVM pero que con MPICH no se realiza. Consiste en poner en marcha la máquina virtual. LAM sí que necesita este proceso. Para ello debemos comenzar el trabajo con el comando “lamboot” que se habrá instalado en el directorio /bin de nuestra ruta. Si no se especifica nada, se incluirán todas las máquinas posibles en el cluster. Si se desea especificar un subconjunto de ellas, se deberá emplear la opción: *lamboot –v maquinas*. Donde maquinas es un fichero de texto con la lista de los nombres de las máquinas que se van a incluir. Después de finalizar el trabajo, se deberá detener la máquina con “lamhalt”.

Otro comando interesante es “lamnodes”, que proporciona una lista de las máquinas incluidas en el cluster una vez arrancado.

Para trabajar con LAM tenemos que ver la forma de compilar y correr los programas. Para compilar se empleará mpicc o mpiCC para programas en c o c++ respectivamente. En este momento pueden aparecer errores por falta de los ficheros de cabecera. Esto es porque el proceso de instalación no tiene muy claro dónde van a parar, ya que considera que deben ir a la misma ruta que los ficheros fuentes, pero ésta se ha tenido que modificar para poder terminar el proceso con éxito. La solución es copiarlos desde el lugar en que han quedado instalados hasta aquel en que los busca. Una vez solucionados estos problemas, se puede proceder a la compilación de las aplicaciones con: *mpicc –c –o ejecutable fuente.c*

Una vez compilada la aplicación, se procede a su ejecución: *mpirun –np numero_de_procesos ejecutable* Esto debería funcionar perfectamente, pero se ha comprobado que puede no ser así. En la versión de LINUX utilizada (SUSE 7.3) ha sido necesario incluir la opción –lamd con mpirun para que las aplicaciones funcionen. Esta opción hace que la comunicación entre procesos se realice a través del demonio “lamd”.

Finalmente, LAM-MPI proporciona un herramienta gráfica que permite monitorizar las aplicaciones que se lanzan contra el cluster. Se trata de XMPI. Esta herramienta se encontraba disponible entre las opciones de instalación de software de LINUX SUSE 7.3 pero, lamentablemente no funciona con LAM 6.5.6, por lo que se ha empleado la última versión beta disponible en este momento en www.lam-mpi.org/beta: XMPI 2.2.3b6. El proceso de instalación es análogo al de todo el software instalado hasta el momento. Solamente se deberá tener en cuenta a la hora de la configuración, incluir como ruta de instalación aquella en la que se encuentran los ejecutables de MPI, por ejemplo: *./configure –prefix=/usr/local/lam-mpi/bin*. En el momento de la instalación pueden aparecer errores por falta de los ficheros de cabecera, aunque si se copiaron todos en el momento de instalar LAM, no habrá ningún problema.

XMPI se debe iniciar desde el entorno gráfico; de lo contrario nos dará un error de violación de segmento.