BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

Procesamiento Paralelo En Un Cluster Bajo Linux Utilizando MPI

Tesis que Para Obtener el Titulo De Licenciado En Ciencias De La Computación Presenta:

Roxana Solís Ramírez

Asesor

Dr. Manuel Martín Ortíz

INDICE

INTRODUCCIÓN

En muchas ramas de la ciencia, debido a la complejidad de los problemas que se estudian se requiere contar con acceso a una máquina que sea capaz de desarrollar varios miles de millones de operaciones por segundo. En los últimos años, el personal académico de diversas universidades y centros de investigación se han dado a la tarea de aprender a construir sus propias supercomputadoras conectando computadoras personales y desarrollando software para enfrentar tales problemas extraordinarios, dando lugar a emplear lo que llamamos clusters; que no es otra cosa más que una colección de computadoras interconectadas de alguna manera, que trabajan en conjunto, distribuyéndose las tareas entre ellas, logrando que el usuario lo vea como una sola; la idea de utilizar un cluster en la resolución de problemas es reducir el tiempo de computación mediante el reparto de la carga entre sus nodos.

La Facultad de Ciencias de la Computación cuenta con un cluster homogéneo con las siguientes características: 16 nodos, donde cada nodo contiene dos procesadores a 2 GHz, 2 GB de RAM y un HD SCSI de 36 GB. La comunidad estudiantil tiene acceso a este para resolver determinados problemas. Ante esto surge la necesidad de saber explotar al máximo sus recursos; mediante lo cual se logrará una mayor flexibilidad en el manejo de la resolución de problemas al que se le someta.

En esta tesis se implementa el desarrollo de tres algoritmos que ilustran las principales características de la programación para aplicaciones paralelas basada en paso de mensajes conocida como MPI con el lenguaje C, bajo linux. Donde los objetivos son:

- Demostrar la utilidad y potencialidad del procesamiento paralelo, utilizando un cluster.
- Mostrar que MPI es un software eficiente para resolver problemas paralelos

Para tal efecto, en esta tesis se realiza:

- Clasificación de los diferentes tipos de paralelismo
- Destacar las características de MPI
- Diseño y Ejecución de algoritmos paralelos
- Presentación de gráficas del tiempo de procesamiento de los algoritmos

En el capítulo uno, se explica el concepto de cluster, arquitectura y clasificación, además de aspectos importantes sobre el paralelismo, así como las arquitecturas y algunos lenguajes de programación utilizados para su implementación.

En el capitulo dos, se documenta el origen, historia y evolución del lenguaje de programación estándar MPI, de las plataformas así como la estructura general de un programa en MPI, el manejo de los datos de entrada y salida, finalmente se presenta un ejemplo de programa usando el clásico ¡hola mundo!.

En el capítulo tres, se presentan tres algoritmos que ponen de relieve las principales características de MPI, a saber, el algoritmo de montecarlo: en el que se muestra como se lleva a cabo el intercambio de paso de mensajes de forma bloqueante y no bloqueante, el segundo es la regla del trapecio, el cual se encarga de calcular la integral de 3 funciones predefinidas en el programa, este nos muestra el potencial de MPI, para llevar a cabo el paso de mensajes de manera más eficaz al emplear el uso de la comunicación colectiva y, como último algoritmo, el cálculo de la multiplicación de matrices de fox, el cual nos pone de relieve las características de MPI para crear sus propios comunicadores, el intercambio de mensajes mediante broadcasts y paso de mensajes simples, así como el uso de topologías.

Finalmente, en el capítulo cuatro se presentan las gráficas de los resultados de pruebas realizadas en el cluster, seguido del análisis de cada uno de los algoritmos en particular, además de una serie de conclusiones basadas en los resultados de los datos graficados.

CAPITULO 1

1. ¿QUÉ ES UN CLUSTER?

Los Clusters en computación son una colección de computadoras (nodos, estaciones de trabajo, o SMP's) interconectadas de alguna manera, que trabajan en conjunto, distribuyéndose las tareas entre ellas, logrando que el usuario lo vea como una sola.

Un nodo del sistema puede ser un sistema con uno o mas procesadores con memoria, recursos de entrada / salida, y un sistema operativo. Estos nodos pueden coexistir en una misma caja, o bien estar físicamente separados y conectados a través de una LAN, lo que puede parecer como un sistema único a los usuarios y aplicaciones.

1.1.1 ¿COMO FUNCIONA UN CLUSTER?

Desde un punto de vista general, un cluster consta de dos partes. La primera es el software: un sistema operativo confeccionado especialmente para esta tarea (por ejemplo un Kernel Linux modificado), compiladores y aplicaciones especiales, que permite que los programas que se ejecutan en el sistema exploten todas las ventajas del cluster. En el entorno de GNU/Linux hay que destacar la PVM(Paralell Virtual Machine) y la MPI(Message Passing Interface), librerías que abstraen la componente hardware de la componente software.

La segunda componente es la interconexión hardware entre las máquinas (nodos) del cluster. Se han desarrollado interfaces de interconexión especiales muy eficientes, pero es común realizar las interconexiones mediante una red Ethernet dedicada de alta velocidad, por medio de la cual los nodos del cluster intercambian entre sí las tareas, las actualizaciones de estado y los datos del programa. En un cluster abierto, existirá una interfaz de red que conecte al cluster con el mundo exterior (Internet).

1.1.2 CONCEPTOS BÁSICOS DE LOS CLUSTERS

Clustering: Implica proveer niveles de disponibilidad y escalabilidad de un sistema al menor costo.

Escalabilidad: es la capacidad de un equipo para hacer frente a volúmenes de trabajo cada vez mayores, sin por ello dejar de prestar un nivel de alto rendimiento aceptable. Existen dos tipos de escalabilidad:

- **Escalabilidad vertical** (también denominada escalabilidad de hardware). Se refiere a cuando tratamos con un gran sistema, con múltiples CPUs, con una gran velocidad de interconexión entre nodos. La mayoría de las aplicaciones escalan muy bien verticalmente hasta cierto número de CPUs. Actualmente, debido a su alto precio sólo se emplea en aplicaciones muy transaccionales y/o minería de datos. El problema de estás máquinas es que suponen una gran inversión que pocas organizaciones pueden afrontar y los problemas de escalabilidad que provocan.

- **Escalabilidad horizontal** (escalabilidad de software). Consiste en emplear muchas máquinas pequeñas e interconectadas para realizar una tarea determinada. Con un costo proporcionalmente bajo por máquina, es una solución barata para algunas aplicaciones como servicios web, correo ó supercomputación. Su principal ventaja es que al crecimiento de un mayor número de nodos se puede hacer cuando sea necesario. Las principales desventajas son que escalan mal para aplicaciones transaccionales (bases de datos, por ejemplo), y que la administración de muchas máquinas interconectadas implican una mayor atención y conocimiento que la administración de una gran máquina con muchos procesadores [7].

La disponibilidad: es la calidad de estar presente, listo para su uso, a mano, accesible [7].
La fiabilidad: es la probabilidad de un funcionamiento correcto [7].

1.1.3 CLASIFICACION DE LOS CLUSTERS

1.1.3.1 POR EL OBJETIVO DE LA APLICACIÓN

1.1.3.1.1 CLUSTERS DE ALTO RENDIMIENTO

Los clusters de alto rendimiento también llamados HP o AR, han sido creados para compartir el recurso más valioso de un ordenador, es decir, el tiempo de proceso. Generalmente se usan en ambientes científicos, o en grandes empresas donde se utilizan para la compilación o renderización.

Cualquier operación que necesite altos tiempos de CPU y millones de operaciones puede ser utilizada en un Cluster de alto rendimiento, siempre que se encuentre un algoritmo que sea paralelizable. Existen Clusters que pueden ser denominados de alto rendimiento tanto a nivel de sistema como a nivel de aplicación. A nivel de sistema tenemos openMosix, mientras que a nivel de aplicación se encuentran otros como MPI, PVM, Beowulf entre otros. En cualquier caso, estos clusters hacen uso de la capacidad de procesamiento que pueden tener varias máquinas [1][2].

1.1.3.1.2 CLUSTERS DE ALTA DISPONIBILIDAD

Definiremos un cluster de alta disponibilidad como un sistema capaz de encubrir los fallos que se producen en él para mantener una prestación de servicio continua. La principal prestación de un sistema de alta disponibilidad es que el fallo de un nodo derive en que las aplicaciones que se ejecutaban en él sean migradas a otro nodo del sistema. Este migrado puede ser automático (failover) o manual (switchover).

Desde un punto de vista general, una solución de alta disponibilidad consiste en dos partes: la infraestructura de alta disponibilidad y los servicios; donde la infraestructura consiste en componentes software que cooperan entre sí para permitir que el cluster aparezca con un único sistema. Sus funciones son: monitorizar los nodos, los procesos de interrelación entre nodos, controlar el acceso a los recursos compartidos y asegurar la integridad de los datos y la satisfacción de los requerimientos del usuario, estas funciones deben estar disponibles cuando el cluster está en un estado estable y aún cuando algunos nodos dejan de estar operativos.

Los servicios de alta disponibilidad son clientes de la infraestructura, y usan las facilidades que exporta ese nivel para mantener en todo momento el servicio a los

usuarios. Los servicios que se mantienen típicamente sobre este tipo de clusters son las bases de datos, sistemas de ficheros, servidores de correo y los servidores Web[1][2].

1.1.3.1.3 CLUSTERS DE ALTA CONFIABILIDAD

Estos clusters tratan de aportar la máxima confiabilidad en un entorno, en el cual se necesite saber que el sistema se va a comportar de una manera determinada.

Este tipo de Clusters son los más difíciles de implementar, generalmente se basan en no solamente conceder servicios de alta disponibilidad, sino en ofrecer, un entorno de sistema altamente confiable. Esto implica en sí miso muchísima sobrecarga en el sistema. Son también clusters muy acoplados [7].

1.1.3.2 POR EL USO DE CADA NODO

1.1.3.2.1 CLUSTERS DEDICADOS

En los clusters dedicados, se utilizan todos los recursos para la computación paralela, es decir, son dedicados a tareas específicas, como por ejemplo consolas o estaciones de monitoreo [1][2].

1.1.3.2.2 CLUSTERS NO DEDICADOS

En los clusters no dedicados, cada ordenador ejecuta sus propios programas y el cluster toma los ciclos en los que cada nodo no realiza alguna tarea y los utiliza para el procesamiento paralelo [1][2].

1.1.3.3 POR EL HARDWARE DE CADA NODO

1.1.3.3.1 CLUSTERS DE PC`s (CoPs)

Un cluster de PC`s es un grupo de equipos independientes que ejecutan una serie de aplicaciones de forma conjunta y aparecen ante clientes y aplicaciones como un solo sistema. Los clusters permiten aumentar la escalabilidad, disponibilidad y fiabilidad de múltiples niveles de red [1][2].

1.1.3.3.2 CLUSTERS DE ESTACIONES DE TRABAJO (COWs)

Un cluster de estaciones de trabajo toma como base la utilización de un grupo de estaciones de trabajo conectadas en red para ejecutar aplicaciones paralelas y distribuidas. De otra forma se puede decir que es un conjunto de PC`s compactos sin teclado ni pantalla ubicados en un mismo mueble.

1.1.3.3.3 CLUSTERS DE SMP`s (CLUMPs)

Los clusters de SMP tienen un fuerte acoplamiento y serán integrados a un cluster por medio de una red de interconexión de baja latencia y gran ancho de banda. Así, se espera que surjan clusters heterogéneos de SMP`s con uno, dos, cuatro u ocho procesadores por nodo [1][2].

1.1.4 APLICACIÓN DE LOS CLUSTERS

Los sistemas cluster han empezado a ser usados en donde los sistemas mainframes o supercomputadoras eran utilizadas en el pasado.

Los clusters han formado parte significativa en el avance de las siguientes industrias.

- Investigación científica y defensa
- Procesos de energía sísmica
- Efectos visuales y entretenimiento
- Internet/ISP/ASP

En algunos puntos específicos

- Identificar nuevas partículas sub-atómicas, cálculos moleculares
- Investigación para descubrir una droga para el SIDA
- Exploración de recursos petróleo, gas
- Investigación del genoma humano

- Aplicaciones en áreas como predicción del clima, astronomía, biología, química, etc. son algunas de las aplicaciones más comunes para los clusters.

- Otro ejemplo de aplicación mas usado en estos días, es por ejemplo, un negocio en Internet que recibe millones de peticiones por día, y nuestro trabajo es asegurarnos de que los servidores respondan rápidamente a las peticiones de los clientes.

1.2 PROCESAMIENTO PARALELO

El procesamiento paralelo consiste en dejar que varias unidades de procesamiento (como computadoras independientes) resuelvan un problema extenso, y ha surgido como una tecnología clave en la computación de nuestros días. Se tienen dos grandes vertientes para la implantación:

1. **Equipos masivamente paralelos** (MPP's por sus siglas en inglés). Actualmente son las computadoras más potentes en el mundo, y pueden combinar desde unos cientos de procesadores hasta miles en un solo gabinete, a través de un bus especial se conectan a varios gigabytes de memoria. El enorme potencial de los MPP's los hace ideales para el tratamiento de problemas computacionales difíciles, como el modelamiento de fenómenos astronómicos, comportamiento climático y diseño de moléculas.

2. **Computación distribuida**. Forzosamente necesita integrar trabajo colaborativo a través de una red de comunicaciones a la que se conectan las computadoras y por medio de la cual cooperan en la tarea asignada.

La computación paralela implica el uso de dos o más procesadores que trabajan colaborativamente en la solución de un problema. El programa que se vaya a ejecutar en forma paralela requerirá ser subprocesado por todos los elementos de cálculo que se vayan a usar en el sistema y finalmente los resultados individuales de cada uno de ellos se recopilan en un solo punto por donde se obtiene la solución deseada.

Para lograr la coordinación entre los elementos de procesos se usan esencialmente dos mecanismos básicos de modelamiento de procesos paralelos:

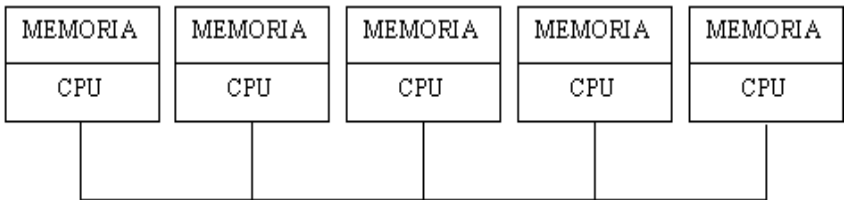

Fig.1.1 Memoria distribuida

1. **Paso de mensajes**. Este mecanismo considera a N procesadores operando en forma independiente uno o diferentes programas, los datos se intercambian entre ellos a través de mensajes (un tipo especial de estructura de datos) conforme se van necesitando para mantener una sincronía y secuenciación con las tareas en ese momento siendo ejecutadas. Cada unidad de proceso puede ser originador o receptor de mensajes en cualquier momento y todas pueden intercambiar datos entre sí. El enrutamiento de mensajes se le deja al sistema operativo, y el número de "saltos" o puntos de intercomunicación por lo que el mensaje debe pasar dependerán de la topología de la máquina en uso. Este sistema puede usarse en una red de estaciones o en grupo de procesadores fuertemente acoplados con memoria distribuida. Las desventajas de este mecanismo son que se requiere agregar bastante código para lograr que un programa sea convertido a usar un esquema y que se puede tener interbloqueo entre los elementos modulares en que se haya descompuesto el programa, al colocar erróneamente las rutinas de pasos de mensajes send y receive sobre las cuales opera.

2. **Memoria compartida**. La idea esencial es identificar las regiones paralela y secuencial de un programa, la región secuencial se ejecuta en un solo procesador y la paralela en múltiples de ellos. La región paralela consiste en múltiples hilos de ejecución cada uno corriendo de forma concurrente. En algunos casos la identificación de las regiones paralela y secuencial puede ser sencillo, pero en otros es necesario indicar al compilador que realice la paralelización automática, lo que hasta el momento es un método poco refinado y requiere intervención del programador para el mejoramiento final del código.

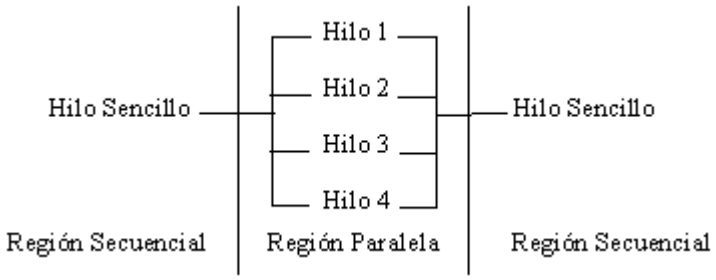

Fig.1.2: Memoria compartida

Los sistemas de memoria compartida pueden desarrollarse y suelen ser más sencillos y más rápidamente que los de paso de mensajes aunque la ganancia en desempeño dependerá del compilador a utilizar. En ambos casos no se puede predecir el desempeño del sistema integrado, aunque el mecanismo de paso de mensajes ofrece mejores datos para su evaluación. Una ventaja grande de paso de mensajes reside en su elevada portabilidad, puesto que por lo general consiste del protocolo de comunicaciones estándar de la red más las rutinas de sincronización y comunicaciones de procesos que se ejecutan sobre dicho protocolo, además de que puede usarse en una red de computadoras heterogénea.

Cabe mencionar que muchos autores hacen la distinción entre los sistemas distribuidos, elaborados para que trabajen con ellos varios usuarios en forma simultánea y los sistemas paralelos, que pretenden lograr la máxima velocidad de ejecución en una tarea determinada. Esta distinción ha tenido mucha controversia, por lo que el consenso general ha determinado usar “sistema distribuido” en sentido amplio, donde varios CPU’s trabajan (intercomunicados entre sí) de manera cooperativa.

La idea fundamental de un sistema distribuido es que constituye una combinación de computadoras y sistemas de transmisión de mensajes bajo un solo punto de vista lógico. Se puede aseverar que el sistema constituye un ente capaz de procesar información debido a las siguientes características:

- El sistema consiste de una cantidad de computadoras cada una de las cuales tiene su propio almacenamiento, dispositivos periféricos y capacidad computacional.

- Todas las computadoras están adecuadamente interconectadas.

Por medio del sistema operativo adecuado, estas computadoras mantienen su capacidad de procesamiento de tarea local, mientras constituyen elementos colaborativos de procesamiento en el ambiente distribuido. Un ambiente distribuido también incluye las siguientes características:

- Una variedad de componentes que incluyen tanto plataformas de cómputo como las redes de interconexión que transportan mensajes entre ellas unificadas en un solo ambiente de procesamiento.

- La transparencia, como resultado de la abstracción apropiada de los componentes del sistema.

Un sistema de computación distribuida es una colección de procesadores interconectados por una red de transferencia de información en la cual cada procesador posee su propio espacio de memoria y otros periféricos. La comunicación entre dos procesadores se lleva a cabo a través de paso de mensajes sobre dicha red. Para un procesador en particular sus recursos son locales, en tanto que otros procesadores y sus recursos son remotos.

Entre las ventajas que ofrecen los sistemas distribuidos con respecto a los sistemas centralizados se encuentran:

- Muchas aplicaciones están elaboradas para operar de forma natural en ambiente disperso tales como: bases de datos, sistemas de trabajo colaborativo y juegos cooperativos o MUD’s.

- Mayor confiabilidad, ya que es factible construir un sistema tolerante a fallas si sus componentes comparten la carga de trabajo y, de presentarse alguna falla, dicha carga es asignada a los demás elementos que siguen operando.

Los sistemas distribuidos con respecto a una sola PC tienen las siguientes ventajas:

- Compartimiento de periféricos/recursos, logrando acceder a equipos especializados que al usarse concurrentemente permiten aplicar criterios económicos para su asignación y adquisición.

- Distribución de carga de las aplicaciones, con lo que se logra mejorar la carga de trabajo de ciertos equipos o servicios, de manera que se balancea adecuadamente el nivel de desempeño y aplicación de componentes de hardware y software, maximizando el uso de los sistemas de acuerdo a la demanda de los usuarios.

Frente a tales ventajas es posible que no se perciban las desventajas, pero al respecto se señalan principalmente tres:

1. El software, en un sistema distribuido el software debe ser de un tipo y capacidad muy especial, hecho específicamente para administrar y operar sobre un ambiente disperso.

2. Las redes de interconexión, a través de ellas fluyen los mensajes y paquetes de comunicación entre los diferentes procesadores involucrados en la tarea y de fallar éstas los procesos asociados pueden dañarse o interrumpirse.

3. La seguridad, si cualquier usuario tiene acceso a través de una imagen lógica a todos los elementos y periféricos que integran un sistema distribuido, también puede leer información o datos de otras personas.

1.3 CLASIFICACIÓN DE MODELOS PARALELOS

1.3.1 CLASIFICACIÓN DE ARQUITECTURAS SEGÚN EL MECANISMO DE CONTROL

En 1966 según algunos, Flynn estableció una clasificación, se basa en dos características:

- Nº de datos que se pueden procesar
- Nº de flujos de instrucciones que se pueden procesar

Según la cual todo sistema de cómputo pertenece a una de las siguientes cuatro categorías:

1. **SISD** (Single Instruction stream Single Data stream) constituye el modelo clásico de arquitectura de computadoras (Von Neumann), en el que más allá de

las variantes, se lleva acabo un procesamiento secuencial mediante la ejecución de una única instrucción sobre un único dato por vez (ej. cargar el contenido de una dirección de memoria en un acumulador, sumar 1 al contenido de un registro, etc.)[7].

2. **SIMD** (Single Instruction stream - Multiple Data stream): aquí se consideran los sistemas que a partir de una única instrucción son capaces de procesar un conjunto o array de datos en forma simultánea, con la particularidad de ejecutar la misma instrucción sobre cada uno de sus elementos. Estos sistemas se componen de varias unidades funcionales idénticas (Unidades Aritmético-Lógicas, Registros de Direcciones de Memoria, etc.) comandadas todas ellas desde una única unidad de control encargada de recibir y procesar cada instrucción a ejecutar. Podría considerarse que, aunque en forma restringida, esta arquitectura incorpora en cierta medida el paralelismo desde el momento que es capaz de ejecutar "algunas tareas" (no "cualquier tarea") en forma simultánea, concretamente puede realizar N operaciones O, sobre N datos D, al mismo tiempo. En rigor no se puede concebir una máquina cuyo modo de operación sea exclusivamente SIMD dado que es imposible pensar en tareas sobre segmentos de datos tales que absolutamente todos tengan aspecto vectorial. Para que esta modalidad pueda ser implementada, la arquitectura correspondiente deberá combinar necesariamente las prestaciones SIMD con las SISD [7].

3. **MISD** (Multiple Instruction stream - Single Data stream)**:** esta categoría corresponde a configuraciones en las que un tren de unidades de procesamiento se surte por uno de sus extremos de un dato extraído de memoria y devuelve un resultado a la misma memoria por el extremo opuesto, en tanto cada unidad del tren toma por entrada al dato procesado por la unidad anterior y da por salida el que será tomado como entrada por la unidad siguiente. La característica MI está en el hecho de que cada unidad de procesamiento va ejecutando una operación distinta sobre un único dato (SD) que va avanzando hasta convertirse en el resultado [7].

4. **MIMD** (Multiple Instruction stream - Multiple Data stream): es un modelo en el cual el sistema puede considerarse de alguna forma particionado en unidades con capacidad de, 1) Desarrollar su propio cómputo y 2) Comunicarse con las

demás; de modo que al ejecutar cada una de ellas un determinado flujo de instrucciones (SI) sobre su propio flujo de datos (SD) se logra que en conjunto, múltiples flujos de instrucciones (MI) actúen sobre otros tantos flujos de datos (MD) [7].

MIMD constituye no sólo la más ambiciosa combinación posible de las dos variables (*Instruction stream* - *Data stream*) sino el fundamento mismo del procesamiento paralelo en su expresión más completa. Estos sistemas a su vez pueden clasificarse en dos grandes categorías:

- **Multiprocesadores** o sistemas de memoria compartida en los cuales la comunicación entre las distintas unidades se realiza, en forma implícita a través de variables compartidas dentro de una única memoria principal

- **Multicomputadores** o sistemas de memoria distribuida en los que la comunicación explícita entre unidades se realiza a través de una Red de paso de mensajes y donde cada unidad de procesamiento opera sobre su propia memoria.

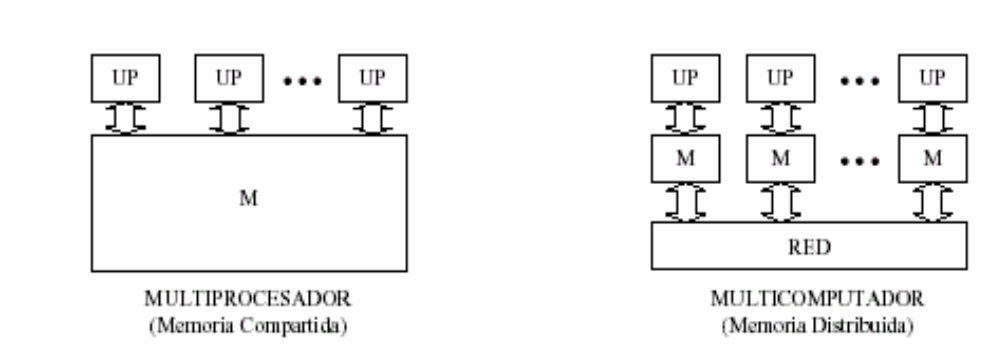

Fig.1.3: Arquitectura de un multiprocesador y multicomputador

1.4 CLASIFICACIÓN EN FUNCIÓN DE LA ORGANIZACIÓN DE LA MEMORIA.

En general, vamos a distinguir tres tipos de organización de la memoria en sistemas multiprocesador. Aunque los sistemas comerciales van a emplear modelos mixtos, esta clasificación sencilla, ayuda suficientemente a comprender las implicaciones que se derivan de cada modelo.

1. UMA (Uniform Memory Access). En estos sistemas la memoria es global para todos los procesadores y los accesos tienen los mismos retardos para todos los nodos. Requiere un hardware sencillo aunque su principal desventaja es el cuello de botella en que se convierte el acceso a memoria y los problemas de escalabilidad que ello supone.

2. NUMA (Non Uniform Memory Access). El espacio de direcciones sigue siendo global pero no así la memoria, que va a ser local a cada procesador aunque accesible por todos los demás. Los accesos tienen retardos no uniformes dependiendo de si el dato buscado se encuentra o no en memoria local. El hardware de la red es complejo ya que se van a necesitar interconexiones especiales que minimicen los retardos en el acceso a memoria remota. Pero, por otra parte, se mejora la escalabilidad.

3. MPM (Message Passing Machines). En este caso la memoria además de pertenecer a cada procesador, va a ser vista mediante un mapa de direccionamiento distribuido. Cada procesador es un computador independiente del resto. Se mejora la escalabilidad notablemente, siendo la red la parte más compleja de escalar. El overhead de comunicación, en cambio, es el más alto. Las arquitecturas MPM son las que hemos venido relacionando con los clusters.

1.5 HERRAMIENTAS PARA EL DESARROLLO DE SOFTWARE PARALELO

1.5.1 LENGUAJES/COMPILADORES

Si bien el lenguaje de programación C es uno de los más potentes para máquinas paralelas, existen lenguajes de más alto nivel adecuados para estos menesteres, entre ellos, cabe citar:

1.5.2 FORTRAN

Fortran, es un lenguaje de alto nivel, con numerosas mejoras con respecto al ANSI del 1966, (objetos como los de C++, instrucciones para procesamiento paralelo,...)
Existen en el mercado compiladores capaces de generar código para arquitecturas tipo SMP y Clusters entre otras. Como ejemplo de estos: HPFC (HPFC Compiler).

1.5.3 MENTAT

Es un lenguaje orientado a objetos para procesamiento en paralelo, funciona en máquinas clusters y se encuentra disponible para Linux. Su sintaxis es similar a la de C++.

1.5.4 mpC PROGRAMMING LANGUAGE

mpC es un lenguaje paralelo de alto nivel (una extensión del C), diseñado especialmente para desarrollar aplicaciones adaptables y portables para redes heterogéneas de ordenadores. La idea principal subyacente en mpC es que cada aplicación define una red abstracta y distribuye datos, computación y comunicaciones sobre la red. El sistema de programación mpC usa esta información para reasignar la red abstracta sobre cualquier red real de forma que se asegura la eficiencia. Este “mapeado” se realiza en tiempo de ejecución y se basa en información acerca de la capacidad de los procesadores y los enlaces de la red real, adaptando dinámicamente el programa a la red.

El sistema mpC encapsula una plataforma particular de comunicación (actualmente, un subconjunto de MPI) asegurando la independencia de la plataforma del resto de componentes del sistema.

1.5.5 CILK

Cilk es un lenguaje para programación paralela multihebrada basado en C. Esta diseñado para utilizarse como un lenguaje de propósito general, pero específicamente efectivo para el paralelismo asíncrono, que puede ser difícil de escribir usando un estilo de paso de mensajes. Cilk utiliza un “scheduler” que permite aproximar la eficiencia de forma precisa.

1.5.6 Jade/SAM (Dialecto concurrente de C) SOBRE PVM

Es un lenguaje de programación paralelo (una extensión de C) para explotar la concurrencia en programas secuenciales de grano grueso.

1.5.7 PVM

PVM ("Parallel Virtual Machine", Máquina Virtual Paralela) es una librería de paso de mensajes libre y portable, generalmente implementada por encima de los sockets. Está claramente establecida como el estándar de paso de mensajes para el procesamiento paralelo en clusters.
PVM puede ser utilizado en monoprocesadores, máquinas SMP, así como clusters conectados por una red que soporte sockets (por ejemplo SLIP, PLIP, Ethernet, ATM, etc). De hecho, PVM funciona incluso en grupos de máquinas con diferentes tipos de procesadores, configuraciones, o diferentes tipos de red (clusters heterogéneos). Puede incluso tratar un grupo bastante amplio de máquinas conectadas a Internet como una sola máquina paralela (una máquina virtual paralela, de ahí su nombre)[7].

El modelo de funcionamiento de PVM es simple pero muy general, y se acomoda a una amplia variedad de estructuras de programas de aplicación. La interfaz de programación es muy sencilla, permitiendo que las estructuras de los programas sencillos sean implementadas de manera intuitiva.

El usuario escribe su aplicación como un conjunto de *tareas* que cooperan. Dichas tareas acceden a los recursos de PVM a través de una librería de funciones con una interfaz estándar. Estas funciones permiten la inicialización y finalización de las tareas a través de la red, así como la comunicación y la sincronización entre dichas tareas. Las operaciones de comunicación utilizan estructuras predefinidas, tanto las encargadas del envío y recepción de datos como aquellas más complejas (sincronización en barrera, suma global, broadcast, etc.).

Debido a su portabilidad, su disponibilidad y su simple pero completa interfaz de programación, el sistema PVM ha ganado una amplia aceptación en la comunicad de cálculo científico de alto rendimiento.

1.5.8 MPI

MPI ("Message Passing Interface", Interfaz de Paso de Mensajes) es un estándar que define la sintaxis y la semántica de las funciones contenidas en una librería de paso de mensajes, diseñada para ser usada en programas que exploten la existencia de múltiples procesadores. Aunque la interfaz no cambie, puede implementarse tanto en sistemas que utilicen paso de mensajes, como sistemas de memoria compartida[7].

La primera versión de MPI fue desarrollada en 1993-1994 por un grupo de investigadores al servicio de la industria, el gobierno y el sector académico. Debido al apoyo recibido MPI es relativamente el nuevo estándar para la programación de procesadores paralelos basado en el paso de mensajes, tomando el testigo de PVM.

Sin embargo existen frecuentes dudas y discusiones por parte de los usuarios acerca de cuál es el estándar a utilizar. Así pues haremos un resumen de las diferencias más características entre los sistemas más utilizados, PVM y MPI:

1.5.8.1 ENTORNO DE EJECUCIÓN

Simplemente PVM *tiene uno definido*, mientras que *MPI no especifica* cómo debe ser implementado. Así el inicio de la ejecución de un programa PVM será realizado de manera idéntica en cualquier plataforma, mientras que para MPI depende de la implementación que estemos utilizando.

1.5.8.2 SOPORTE PARA CLUSTERS HETEROGÉNEOS

PVM fue desarrollado para aprovechar ciclos de CPU de estaciones de trabajo ociosas, por lo que maneja de manera directa mezclas heterogéneas de máquinas y sistemas operativos. En contraste MPI asume que su principal objetivo son los MPPs y los clusters dedicados de estaciones de trabajo casi idénticas. Sin embargo, dependiendo de la implementación que utilicemos, tendremos un soporte más o menos adecuado.

1.5.8.2.1 AMPLITUD DEL CAMPO DE ESTUDIO:

PVM evidencia una unidad de propósito que MPI no tiene. Como veremos la especificación MPI-1 centra su atención en el modelo de paso de mensajes, al igual que PVM. Sin embargo la especificación MPI-2 incluye muchas características que van más allá del modelo de paso de mensajes, como el acceso paralelo a ficheros de E/S o el acceso a memoria remota, entre otros muchos ejemplos.

1.5.8.2.2 DISEÑO DE INTERFAZ DE USUARIO:

MPI fue diseñado después de PVM, y claramente aprendió de él. MPI ofrece una interfaz más clara y eficiente, con manejo de buffers y abstracciones de alto nivel que permiten definir las estructuras de datos a ser transmitidas por los mensajes.

1.5.8.2.3 IMPORTANCIA DEL ESTÁNDAR:

El hecho de que MPI sea respaldado por un estándar formal ampliamente apoyado significa que el uso de MPI es, en muchas instituciones, cuestión de política. MPI no está concebido para ser una infraestructura software aislado y autosuficiente para ser usada en procesamiento distribuido. MPI no necesita planificación de procesos, herramientas de configuración de la plataforma a usar, ni soporte para E/S. Como resultado MPI es implementado normalmente como interfaz de comunicaciones, utilizando las facilidades ofrecidas por el sistema que vayamos a usar, tal como se indica en la figura 1.3 (comunicación vía sockets, operaciones de memoria compartida, etc).

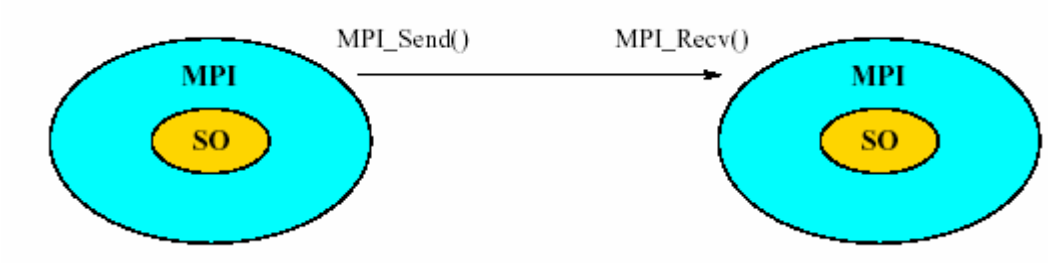

Fig. 1.3: Comunicación de MPI

Éste escenario es ideal para que los programas PVM sean portados a MPI, de manera que se aproveche el alto rendimiento de comunicación que ofrecen algunas compañías en sus arquitecturas.

Dado que éste es el estándar en el que nos basaremos para el desarrollo del presente documento, en el capítulo 2 haremos una descripción más elaborada de sus características.

1.5.9 P4

El sistema p4 es una librería de macros y funciones desarrollada por el Argonne Nacional Laboratory 1 destinadas a la programación de una gran variedad de máquinas paralelas. El sistema p4 soporta tanto el modelo de memoria compartida (basado en monitores) como el modelo de memoria distribuida (basado en paso de mensajes).

Para el modelo de memoria compartida, p4 proporciona un conjunto de monitores así como funciones desarrolladas para crearlos y manipularlos; un monitor es un mecanismo de sincronización de procesos en sistemas con memoria compartida. Para el modelo de memoria distribuida, p4 proporciona operaciones de envío y recepción, y un método de administración de procesos basado en un fichero de configuración que describe la estructura de los grupos y procesos.
Dicho fichero especifica el nombre del servidor, el fichero que será ejecutado en cada máquina, el número de procesos que serán iniciados en cada servidor (principalmente para sistemas multiprocesador) e información auxiliar. Un ejemplo de fichero de configuración es:

```
# start one eslave on each of sun2 and sun3
local 0
sun2 1 /home/mylogin/pupgms/sr_test
sun3 1 /home/mylogin/pupgms/sr_test
```

Dos cosas son notables en lo que se refiere al mecanismo de administración de los procesos en p4:

- Primero, existe la noción de procesos *maestros* y procesos *esclavos*, y pueden formarse jerarquías multinivel para implementar el denominado modelo de cluster de procesamiento.

- Segundo, el principal modo de creación de procesos es estático, a través del fichero de configuración; la creación dinámica de procesos es posible sólo mediante procesos creados estáticamente que deben invocar una función p4 que expande un nuevo proceso en la máquina local. A pesar de estas restricciones una gran variedad de paradigmas de aplicación pueden ser implementados en el sistema p4 de una manera bastante sencilla.

El paso de mensajes es conseguido en el sistema p4 a través del uso de primitivas **send** y **receive** tradicionales, con casi los mismos parámetros que los demás sistemas de paso de mensajes. Algunas variantes han sido proporcionadas por cuestiones semánticas, como el intercambio heterogéneo y las transferencias bloqueantes o no bloqueantes. Sin embargo una proporción significativa de la administración del buffer y su carga se le deja al usuario. Aparte del paso de mensajes básico, p4 ofrece también una variedad de operaciones globales incluyendo broadcast, máximo y mínimo globales, y sincronización por barrera.

1.5.10 EXPRESS

En contraste con lo que sucede con otros sistemas de procesamiento paralelo descritos en esta sección, Express es una colección de herramientas que individualmente dirigen varios aspectos del procesamiento concurrente. Dicho paquete software es desarrollado y comercializado por Parasoft Corporation, una compañía creada por algunos miembros del proyecto de procesamiento concurrente de Caltech.

La filosofía de Express se basa en comenzar con la versión secuencial de un algoritmo y aplicarle un ciclo de desarrollo recomendado, para así llegar a la versión paralela de dicho algoritmo de manera óptima. Los ciclos de desarrollo típicos comienzan con el uso de la herramienta VTOOL, una aplicación gráfica que permite mostrar por pantalla el progreso de los algoritmos secuenciales dinámicamente. Puede mostrar actualizaciones y referencias a estructuras de datos individuales, con la intención de

demostrar la estructura de los algoritmos y proporcionar la información necesaria para su paralelización.

En relación con este programa tenemos la herramienta FTOOL, que proporciona un análisis en profundidad de los programas. Dicho análisis incluye información acerca del uso de variables, la estructura de flujo de control, etc. FTOOL opera tanto en las versiones secuenciales de los algoritmos como en las paralelas. Una tercera herramienta llamada ASPAR es usada entonces; esta herramienta es un paralelizador automático que convierte programas secuenciales escritos en C o Fortran en sus versiones paralelas o distribuidas, usando los modelos de programación Express.

El núcleo del sistema Express es un conjunto de librerías de comunicación, E/S y gráficos paralelos. Las primitivas de comunicación son parecidas a las encontradas en otros sistemas de paso de mensajes e incluyen una variedad de primitivas para realizar operaciones globales y de distribución de datos. Las funciones de E/S extendida permiten la E/S paralela. Un conjunto de funciones similar es proporcionado para mostrar gráficos de múltiples procesos concurrentes. Express contiene además la herramienta NDB, un depurador paralelo que utiliza comandos basados en la popular interfaz **dbx**.

1.5.11. LINDA

Linda es un modelo concurrente de programación fruto de la evolución de un proyecto de investigación de la Universidad de Yale. El concepto principal en Linda es el *espacio de tuplas*, una abstracción a través de la cual se comunican los procesos. Este tema central de Linda ha sido propuesto como paradigma alternativo a los dos métodos tradicionales de procesamiento paralelo: el basado en memoria compartida, y el basado en paso de mensajes. El espacio de tuplas es esencialmente una abstracción de la memoria compartida/distribuida, con una diferencia importante: los espacios de tuplas son asociativos. También existen otras distinciones menores. Las aplicaciones utilizan el modelo Linda introduciendo en los programas secuenciales estructuras que manipulan el espacio de tuplas.

Desde el punto de vista de las aplicaciones, Linda es un conjunto de extensiones para lenguajes de programación que facilitan la programación paralela. Proporciona una

abstracción de la memoria compartida que no requiere la existencia de hardware específico que comparta memoria físicamente.

El término Linda se refiere a menudo a implementaciones específicas software que soportan el modelo de programación Linda. Este tipo de software establece y mantiene espacios de tuplas, siendo utilizado en conjunción con librerías que interpretan y ejecutan primitivas Linda. Dependiendo del entorno (mutiprocesadores con memoria compartida, sistemas de paso de mensajes, etc.) el mecanismo de espacio de tuplas será implementado utilizando diferentes técnicas con varios grados de eficiencia.

Recientemente ha sido propuesto un nuevo esquema relacionado con el proyecto Linda. Este esquema, denominado *Pirhana*, propone un planteamiento revolucionario en el procesamiento concurrente: los recursos computacionales (vistos como agentes activos) eligen a los procesos basándose en su disponibilidad y conveniencia. Este esquema puede ser implementado en múltiples plataformas y es conocido como Sistema Pirhana o Sistema Linda-Pirhana.

CAPITULO 2

EL ESTÁNDAR MPI

INTRODUCCIÓN

MPI ("Message Passing Interface", Interfaz de Paso de Mensajes) es un estándar que define la sintaxis y la semántica de las 129 funciones contenidas en una librería de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.

Dicha librería puede ser utilizada por una amplia variedad de usuarios para implementar programas que utilicen paso de mensajes.
Desde la finalización de su primera versión en Junio de 1994, MPI ha sido ampliamente aceptado y usado. Existen implementaciones muy estables y eficientes, incluyendo algunas de dominio público.

2.1 ORIGEN

El paso de mensajes es un paradigma de programación ampliamente utilizado en computadores paralelos, especialmente en aquellas con memoria distribuida. Aunque existen muchas variantes, el concepto básico de procesos comunicándose mediante mensajes está muy consolidado.
Durante los años anteriores a la aparición del estándar MPI se percibía un progreso sustancial en proyectar aplicaciones importantes en este paradigma. De hecho, cada compañía había implementado ya su propia variante. Más recientemente algunos sistemas de dominio público habían demostrado que el paso de mensajes podía implementarse eficientemente de manera portable [9].

Fig. 2.1 Interfaz de paso de mensajes

Por todo ello comprendemos que éste era el momento apropiado para definir tanto la sintaxis como la semántica de una librería estándar de funciones que fuera útil a una amplia variedad de usuarios y que pudiera implementarse eficientemente en una amplia gama de computadores. Y de esa idea nació el estándar MPI.

Los diseñadores de MPI pretendieron incorporar la mayoría de las características más atractivas de los sistemas existentes de paso de mensajes, antes que seleccionar uno de ellos y adoptarlo como el estándar. Así, en su origen MPI estuvo muy influenciado por el trabajo del Centro de Investigación T.J.Watson de IBM, el sistema NX/2 de Intel, Express, Vertex de nCUBE y PARMACS. Otras contribuciones importantes fueron las realizadas por Zipcode, Chimp, PVM, Chameleon y PICL.

Los diseñadores de MPI identificaron algunos defectos críticos de los sistemas de paso de mensajes existentes en áreas como la composición de datos complejos, la modularidad y las comunicaciones seguras. Esto permitió la introducción de nuevas características en MPI [9].

2.1.1 HISTORIA

El esfuerzo de estandarización de MPI involucró a unas 60 personas procedentes de 40 organizaciones, principalmente de EEUU y Europa. La mayoría de las principales compañías de computadores paralelos del momento estuvieron involucradas en MPI, así como investigadores provenientes de universidades, laboratorios de gobierno y la industria. El proceso de estandarización comenzó con el Seminario sobre Estándares de Paso de Mensajes en Entornos de Memoria Distribuida, patrocinado por el Centro de Investigaciones sobre Procesamiento Paralelo, mantenido durante los días 29-30 de Abril de 1992 en Williansburg, Virginia (EEUU). En este seminario se discutieron las características esenciales que debía tener una interfaz estándar de paso de mensajes y se estableció un grupo de trabajo para continuar con el proceso de estandarización.

Un borrador preliminar, conocido como MPI1, fue propuesto por Dongarra, Hempel, Hey y Walker en Noviembre de 1992 y una versión revisada fue completada en Febrero de 1993.

MPI1 abarcaba las características principales que fueron identificadas en el seminario de Williansburg como necesarias en un estándar de paso de mensajes. Dado que fue generado para promover la discusión, este primer borrador se centró principalmente en las comunicaciones punto-a-punto. Aunque tocaba muchos asuntos referentes a la estandarización, no incluía operaciones colectivas ni tenía en cuenta la utilización de hilos.

En Noviembre de 1992 una reunión del grupo de trabajo de MPI tuvo lugar en Minneapolis, en la cual se decidió hacer el proceso de estandarización de una manera más formal, adoptando la organización y los procedimientos del "High Performance Fortran Forum" (Foro Fortran de Alto Rendimiento). Fueron formados subcomités para cada una de las principales áreas que componen el estándar, y se estableció un foro de discusión vía e-mail para cada una de ellas. De esta manera quedó conformado el **Foro MPI**, a cuyo cargo quedaría el proceso de estandarización de MPI.

En dicha reunión se propuso el objetivo de presentar un borrador del estándar MPI en el otoño de 1993. Para conseguirlo el Foro MPI mantuvo reuniones cada 6 semanas durante los primeros 9 meses de 1993, presentando el borrador del estándar MPI en la conferencia Supercomputing '93 en Noviembre de 1993. Después de un período de comentarios públicos, que resultaron en algunos cambios en MPI, la versión 1.0 de MPI fue presentada en Junio de 1994.

A principios de Marzo de 1995 el Foro MPI empezó a reunirse de nuevo para considerar correcciones y extensiones al documento original del estándar MPI. El primer producto fruto de estas deliberaciones fue la versión 1.1 de MPI, presentada en Junio de 1995.

En Julio de 1997 aparecen al mismo tiempo la versión 1.2 de MPI y la especificación MPI-2. La versión 1.2 de MPI contiene correcciones y clarificaciones referentes a la versión 1.1. Sin embargo MPI-2 añade nuevos elementos al estándar MPI-1, definiendo la versión 2.0 de MPI.

Todavía hoy continúan las discusiones acerca de las áreas hacia las cuales podría ser útil expandir el estándar MPI; sin embargo en muchas de ellas es necesario obtener más experiencia y realizar más discusiones. De todo ello se encarga un documento separado, el MPI "Journal Of Development" (Periódico de Desarrollo), que no forma parte de la

especificación MPI-2. Y por supuesto el Foro MPI sigue abierto a las aportaciones que puedan realizar los usuarios del estándar, con el objetivo de mantenerlo y desarrollarlo [9].

2.1.2 OBJETIVOS

La finalidad de MPI, de manera concisa, es desarrollar un estándar para escribir programas de paso de mensajes que sea ampliamente utilizado. Para ello la interfaz debe establecer un estándar práctico, portable, eficiente, escalable, formal y flexible.

2.1.3 OBJETIVOS PRINCIPALES

2.1.3.1 PORTABILIDAD

El principal objetivo de MPI, es conseguir un alto grado de portabilidad entre diferentes máquinas. De esta manera el mismo código fuente de paso de mensajes debería poder ser ejecutado en una variedad de máquinas tan grande como la soportada por las distintas implementaciones de MPI, aunque pueda ser necesaria una puesta a punto para obtener la máxima ventaja de las características de cada sistema. A pesar de que el paso de mensajes muchas veces es considerado algo propio de los sistemas paralelos de memoria distribuida, el mismo código podría ser ejecutado en un sistema paralelo de memoria compartida. Puede ejecutarse en clusters o incluso en un conjunto de procesos ejecutándose en una misma máquina lo que nos da cierto grado de flexibilidad en el desarrollo del código, la búsqueda de errores y la elección de la plataforma que finalmente utilizaremos [8] [9].

2.1.3.2 HETEROGENEIDAD

Otro tipo de compatibilidad ofrecido por MPI es su capacidad para ejecutarse en sistemas heterogéneos de manera transparente. Así pues, una implementación MPI debe ser capaz de extender una colección de procesos sobre un conjunto de sistemas con arquitecturas diferentes, de manera que proporcione un modelo de computador virtual que oculte las diferencias en las arquitecturas. De este modo el usuario no se tiene que preocupar de si el código está enviando mensajes entre procesadores de la misma o distinta arquitectura. La implementación MPI hará automáticamente cualquier

conversión que sea necesaria y utilizará el protocolo de comunicación adecuado. Sin embargo MPI no prohíbe implementaciones destinadas a un único y homogéneo sistema, así como tampoco ordena que distintas implementaciones MPI deban tener la capacidad de interoperar. En definitiva, los usuarios que quieran ejecutar sus programas en sistemas heterogéneos deberán utilizar implementaciones MPI diseñadas para soportar heterogeneidad [9].

2.1.3.3 RENDIMIENTO

La portabilidad es un factor importante, pero el estándar no conseguiría una amplia utilización si se consiguiera dicha portabilidad a expensas del rendimiento. Por ejemplo, el lenguaje Fortran es comúnmente usado por encima de los lenguajes ensambladores porque los compiladores casi siempre ofrecen un rendimiento aceptable comparado con la alternativa no portable que representa el lenguaje ensamblador. Un punto crucial es que MPI fue cuidadosamente diseñado de manera que permite implementaciones eficientes. Las elecciones en el diseño parecen haber sido hechas correctamente, dado que las implementaciones MPI están alcanzando un alto rendimiento en una amplia variedad de plataformas; de hecho el rendimiento alcanzado en dichas implementaciones es comparable al de los sistemas presentados por las compañías, los cuales están diseñados para arquitecturas específicas y tienen menor capacidad de portabilidad [9].

Un objetivo importante de diseño en MPI fue el de permitir implementaciones eficientes para máquinas de diferentes características. Por ejemplo, MPI evita cuidadosamente especificar la manera en que las operaciones tienen lugar. Sólo especifica qué hace una operación lógicamente. Como resultado MPI puede ser fácilmente implementado en sistemas que tienen buffer de mensajes en el proceso emisor, en el receptor, o que no tienen buffers para nada. Las implementaciones pueden beneficiarse de las ventajas que ofrecen los subsistemas de comunicación de varias máquinas a través de sus características específicas. En máquinas con coprocesadores de comunicación inteligentes podemos cargar sobre dichos coprocesadores la mayoría del procesamiento relativo al protocolo de paso de mensajes. En otros sistemas la mayoría del código de comunicación será ejecutada por el procesador principal.

Otro ejemplo es el uso de *objetos opacos* en MPI; por ejemplo los elementos grupo y comunicador son objetos opacos. Desde un punto de vista práctico esto significa que los detalles de su representación interna dependen de la implementación MPI particular, y como consecuencia el usuario no puede acceder directamente a ellos. En vez de ello el usuario accede a un *manejador* que referencia al objeto opaco, de manera que dichos objetos opacos son manipulados por funciones MPI especiales. De esta manera cada implementación es libre de hacer lo mejor en determinadas circunstancias. Otra elección de diseño importante para la eficiencia es la manera de evitar el trabajo innecesario. MPI ha sido cuidadosamente diseñado de modo que no sea necesaria demasiada información extra con cada mensaje, ni complejas codificaciones o decodificaciones sobre las cabeceras de dichos mensajes. MPI también evita computaciones extra o tests en funciones críticas que degraden el rendimiento. Otra manera de minimizar el trabajo es reducir la repetición de computaciones previas. MPI proporciona esta capacidad a través de construcciones como peticiones de comunicación persistente o los atributos de los comunicadores. El diseño de MPI evita la necesidad de operaciones extra de copia o almacenamiento sobre los datos: en la mayoría de los casos los datos son llevados directamente de la memoria de usuario a la red, y son recibidos directamente de la red a la memoria receptora.

MPI ha sido diseñado para reducir la sobrecarga en la comunicación producida por el procesamiento, utilizando agentes de comunicación inteligentes y ocultando latencias en la comunicación. Esto se ha logrado usando llamadas no bloqueantes, que separan el inicio de la comunicación de su final.

2.1.3.4 ESCALABILIDAD

La escalabilidad es otro objetivo importante del procesamiento paralelo. La escalabilidad de un sistema es su capacidad para responder a cargas de trabajo crecientes. De este modo los programas MPI deben mantener su nivel de rendimiento aunque incrementemos el número de procesos a ejecutar.
MPI permite o soporta la escalabilidad a través de algunas de sus características de diseño. Por ejemplo, una aplicación puede crear subgrupos de procesos que, en turnos, puedan ejecutar operaciones de comunicación colectiva para limitar el número de procesos involucrados.

2.1.3.5 FORMALIDAD

MPI, como todos los buenos estándares, es valioso debido a que define el comportamiento de las implementaciones de manera concisa. Esta característica libera al programador de tener que preocuparse de aquellos problemas que puedan aparecer. Un ejemplo de ello es la garantía de seguridad en la transmisión de mensajes que ofrece MPI. Gracias a esta característica el usuario no necesita comprobar si los mensajes son recibidos correctamente.

2.1.4 RESUMEN DE OBJETIVOS

- Diseñar una interfaz para la programación de aplicaciones.
- La interfaz debe ser diseñada para que pueda ser implementada en la mayoría de las plataformas, sin necesidad de cambios importantes en la comunicación o el software del sistema.
- Permitir implementaciones que puedan ser utilizadas en entornos heterogéneos.
- Permitir una comunicación eficiente.
- Asumir una interfaz de comunicación fiable: el usuario no debe preocuparse por los fallos en la comunicación.
- La semántica de la interfaz debe ser independiente del lenguaje.
- La interfaz debe permitir la utilización de hilos.
- Proporcionar extensiones que añadan más flexibilidad.

2.1.5 USUARIOS

El estándar MPI está pensado para ser utilizado por todo aquel que pretenda desarrollar programas de paso de mensajes codificados en Fortran 77, C y C++. Esto incluye programadores de aplicaciones individuales, desarrolladores de software para máquinas paralelas y creadores de entornos y herramientas.

2.1.6 PLATAFORMAS

El atractivo del paradigma de paso de mensajes proviene al menos en parte de su portabilidad.

Los programas expresados de esta manera podrían ser ejecutados en multicomputadores, multiprocesadores o combinaciones de ambos. Además es posible realizar implementaciones basadas en memoria compartida. El paradigma no queda obsoleto al combinar arquitecturas de memoria distribuida con arquitecturas de memoria compartida, o al incrementar la velocidad de la red. Debe ser a la vez posible y útil implementar dicho estándar en una gran variedad de máquinas, incluyendo dichas "máquinas" que se componen de colecciones de otras máquinas, paralelas o no, conectadas por una red de comunicaciones.

La interfaz es apropiada para su uso en programas MIMD y SPMD. Aunque no proporciona un soporte explícito para la ejecución de hilos la interfaz ha sido diseñada de manera que no perjudique su uso.

MPI proporciona muchas características encaminadas a mejorar el rendimiento en computadores paralelos con hardware de comunicación especializado entre procesadores. De este modo es posible realizar implementaciones nativas MPI de alto rendimiento para este tipo de máquinas. Al mismo tiempo existen implementaciones MPI que utilizan los protocolos estándares de comunicación entre procesadores de Unix, las cuales proporcionan portabilidad a los clusters y redes heterogéneas.

2.1.7 VERSIONES

El estándar MPI se divide básicamente en dos especificaciones, MPI-1 y MPI-2. La siguiente clasificación muestra el nivel de compatibilidad con MPI que posee una implementación dada:

- Mantener compatibilidad con la especificación MPI-1 significa ser compatible con la versión 1.2 de MPI.
- Mantener compatibilidad con la especificación MPI-2 significa proporcionar toda la funcionalidad definida por la especificación MPI-2.
- En todo caso la compatibilidad hacia atrás está preservada. De esta manera un programa MPI-1.1 válido será un programa MPI-1.2 válido y un programa MPI-2 válido; un programa MPI-1.2 válido será un programa MPI-2 válido.

2.1.8 MPI-1

Las versiones 1.0, 1.1 y 1.2 del estándar MPI se engloban en la especificación MPI-1. Dicha especificación centra su atención en el modelo de paso de mensajes. A continuación mostramos una lista con los elementos contenidos en dicha especificación, y aquellos que quedan fuera de ella.

Elementos Incluidos

- Comunicaciones punto a punto
- Operaciones colectivas
- Grupos de procesos
- Contextos de comunicación
- Topologías de procesos
- Administración del entorno
- Enlaces con Fortran 77 y C
- Interfaz para la creación de perfiles de ejecución

Elementos No Incluidos

- Operaciones de memoria compartida
- Operaciones ya soportadas por el sistema operativo de manera estandarizada durante la adopción de MPI; por ejemplo, manejadores de interrupción o ejecución remota
- Herramientas para la construcción de programas
- Facilidades para la depuración de errores
- Soporte específico para hilos
- Soporte para planificación y creación de procesos
- Funciones de E/S paralela

Como vemos MPI-1 está diseñado para aprovechar todas las facilidades ofrecidas por el sistema que vayamos a usar, tanto aquellas pertenecientes al sistema de comunicación (operaciones de memoria compartida, comunicación vía sockets, etc.) como las relativas al entorno de programación (herramientas para la construcción de programas, facilidades para la depuración de errores, etc.). Como resultado MPI es implementado

normalmente como interfaz de comunicaciones, utilizando dichas facilidades ofrecidas por el sistema.

Existen muchas características que fueron consideradas y no se incluyeron en MPI-1. Esto ocurrió por algunas razones: las restricciones de tiempo que se propuso el Foro MPI en acabar la especificación; el sentimiento de no tener suficiente experiencia en algunos de los campos; y la preocupación de que el añadir más características podría retrasar la aparición de implementaciones. De todas maneras las características no incluidas siempre pueden ser añadidas como extensiones en implementaciones específicas, como es el caso de la extensión MPE dentro de la implementación MPICH.

2.1.9 MPI-2

La especificación MPI-2 es básicamente una extensión a MPI-1 que añade nuevos elementos al estándar. La versión 2.0 de MPI pertenece a la especificación MPI-2. Entre las nuevas funcionalidades que se añaden destacan las siguientes:

- Administración y Creación de Procesos
- Comunicaciones Unilaterales
- E/S Paralela
- Operaciones Colectivas Extendidas
- Enlaces con Fortran 90 y C++

La razón por la cual se creó la especificación MPI-2 fue la demanda por parte de los usuarios y desarrolladores de nuevas características en el estándar. De todos modos MPI-2 sólo añade nuevos elementos a MPI-1, pero no lo modifica [8] [9].

2.1.10 IMPLEMENTACIONES

Desde que se completó la primera versión del estándar en 1994 un gran número de implementaciones MPI han sido puestas a disposición de los usuarios. Esto incluye tanto algunas implementaciones portables e independientes como aquellas que han sido desarrolladas y optimizadas por las principales compañías de computadores paralelos. La alta calidad de dichas implementaciones ha sido un punto clave en el éxito de MPI.
A continuación destacamos las tres implementaciones de dominio público más importantes que pueden ser utilizadas en clusters de sistemas Linux:

- MPICH es sin duda la implementación más importante de MPI. La primera versión de MPICH fue escrita durante el proceso de estandarización de MPI, siendo finalmente presentada al mismo tiempo que la versión 1.0 de MPI. De hecho las experiencias de los autores de MPICH se convirtieron en una gran ayuda para el Foro MPI en el proceso de desarrollo del estándar. Su portabilidad es enorme y su rendimiento elevado. Posee compatibilidad total con MPI-1 e implementa muchos elementos de MPI-2.

- LAM fue desarrollada originalmente en el Centro de Supercómputo de Ohio antes de que el estándar MPI fuera presentado. Cuando MPI apareció, LAM adoptó el estándar. LAM no sólo consiste en la librería MPI, si no que además contiene herramientas de depuración y monitorización. Ha sido optimizada para funcionar con clusters heterogéneos de sistemas Unix; sin embargo es muy portable, siendo utilizada en una amplia variedad de plataformas Unix, desde estaciones de trabajo a supercomputadores. LAM posee compatibilidad total con MPI-1 e implementa muchos elementos de MPI-2.

- CHIMP fue desarrollada en el Centro de Cómputo Paralelo de Edimburgo. Como LAM, CHIMP comenzó como una infraestructura independiente de paso de mensajes que luego evolucionó hacia una implementación MPI. CHIMP es conocida principalmente por haber sido utilizada como versión optimizada de MPI para los CRAY T3D y T3E. CHIMP es portable, pudiendo ser utilizada en estaciones de trabajo de Sun, SGI, DEC, IBM y HP, en plataformas Meiko y en el Fujitsu AP 1000.

2.2 ASPECTOS DE UN PROGRAMA MPI BÁSICO

Todos los programas escritos en MPI deben contener la directiva de preprocesador

#include "mpi.h"

El fichero 'mpi.h' contiene las definiciones, macros y prototipos de función necesarios para compilar los programas MPI.

Antes de llamar a cualquier otra función MPI debemos hacer una llamada a **MPI_Init();** la cual debe ser llamada una vez. Sus argumentos son punteros a los parámetros de la función **main(), aargc y argv**. Dicha función permite al sistema hacer todas las configuraciones necesarias para que la librería MPI pueda ser usada. Después de que el programa haya acabado de utilizar la librería MPI, debemos hacer una llamada a **MPI_Finalize().** Esta función limpia todos los trabajos no finalizados (por ejemplo, envíos pendientes que no hayan sido completados, etc.). Así un programa típico MPI, tiene la siguiente composición:

```
…...
.# include “mpi.h”
..
main(int argc, char** argv){
        .
        /*Ninguna llamada a función MPI anterior a esta */
        MPI_Init(&argc, &argv);
        .
        MPI_Finalize();
        /*Ninguna llamada a función MPI posterior a esta*/
        .       .
} /*main*/
```

2.2.1 ALGORITMO ¡Hola Mundo!

Como vimos anteriormente, todos los programas MPI comparten una serie de características. La inicialización y finalización del entorno de ejecución, el uso de funciones para informarnos de dónde está ubicado cada proceso y el momento temporal en que nos encontramos; para ejemplificar todo esto, implementaremos un sencillo programa que envía un mensaje de saludo e indica el número de procesos en ejecución.

En MPI los procesos implicados en la ejecución de un programa paralelo se identifican por una secuencia de enteros no negativos. Si hay *p* procesos ejecutando un programa, éstos tendrán los identificadores 0,1,...,p-1. El siguiente programa hace que el proceso 0 imprima un mensaje de saludo e informe del número de procesos en ejecución, así como del tiempo empleado en el procesamiento.

Los detalles sobre la compilación y ejecución de este programa dependen del sistema que estemos usando.
Cuando el programa es compilado y ejecutado con dos procesos, la salida debería ser de la siguiente manera:

Proceso 0 en 6.953356E-310 Encargado de la E/S

Hello, world!
Numero de Procesos: 2
Tiempo de Procesamiento: 0.000007

Si lo ejecutamos con cuatro procesos la salida sería:

Proceso 0 en 6.953356E-310 Encargado de la E/S

Hello, world!
Numero de Procesos: 4
Tiempo de Procesamiento: 0.000008

Aunque los detalles de qué ocurre cuando el programa es ejecutado varían de una máquina a otra, lo esencial es lo mismo en todos los sistemas, a condición de que ejecutemos un proceso en cada procesador.

- El usuario manda una directiva al sistema operativo que crea una copia del programa ejecutable en cada procesador.

- Cada procesador comienza la ejecución de su copia del ejecutable.

- Diferentes procesos pueden ejecutar diferentes instrucciones bifurcando el programa a través de condicionantes. Dichos condicionantes suelen basarse, como veremos, en el identificador del proceso.

2.2.2 INFORMANDONOS DEL RESTO DEL MUNDO

MPI ofrece la función **MPI_Comm_rank(),** la cual retorna el identificador de un proceso en su segundo argumento. Su sintaxis es:

```
int MPI_Comm_rank(MPI_Comm comunicador, int* identificador )
```

El primer argumento es el *comunicador*. Esencialmente un comunicador es una colección de procesos que pueden enviarse mensaje entre sí. Normalmente para diseñar programas básicos el único comunicador que necesitaremos será **MPI_COMM_WORLD**. Está predefinido en **MPI** y consiste en todos los procesos que se ejecutan cuando el programa comienza [4].

Muchas de las construcciones que empleamos en nuestros programas dependen también del número de procesos que se ejecutan. MPI ofrece la función **MPI_Comm_Size()** para determinar dicho número de procesos. Su primer argumento es el comunicador. Su sintaxis es:

```
int MPI_Comm_size(MPI_Comm comunicador, int* numprocs)
```

2.2.3 EL PROBLEMA DE LA Entrada/Salida

En nuestros algoritmos asumimos que el proceso 0 puede escribir la salida estándar (la ventana del terminal). Prácticamente todos los procesadores paralelos proporcionan este método de E/S. De hecho la mayoría de ellos permiten a todos los procesos tanto leer de la entrada estándar como escribir en la salida estándar. Sin embargo la dificultad aparece cuando varios procesos están intentando ejecutar operaciones de E/S simultáneamente. Para comprender esto expondremos un ejemplo.

Supongamos que diseñamos un programa de manera que cada proceso intente leer los valores *a, b y c* añadiendo la siguiente instrucción:

```
scanf("%d%d%d", &a, &b, &c);
```

Además supongamos que ejecutamos el programa con dos procesos y que el usuario introduce: 10 20 30
¿Qué ocurre? ¿Obtienen ambos procesos los datos? ¿Lo hace uno sólo? O lo que es peor, ¿obtiene el proceso 0 el número 10 y 20 mientras que el proceso 1 obtiene el 30? Si todos los procesos obtienen datos, ¿qué ocurre cuando escribimos un programa en el que queremos que el proceso 0 obtenga el primer valor de entrada, el proceso 1 el segundo, etc.? Y si sólo un proceso obtiene los datos, ¿qué le ocurre a los demás? ¿Es razonable hacer que múltiples procesos lean de una sola terminal?
Por otra parte, ¿qué ocurre si varios procesos intentan escribir datos en la terminal simultáneamente? ¿Se imprimirán antes los datos del proceso 0, después los del proceso 1, y así sucesivamente? ¿O se imprimirán dichos datos aleatoriamente? ¿O, lo que es peor, se mostrarán los datos de los distintos procesos todos mezclados (una línea del proceso 0, dos caracteres del proceso 1, 3 caracteres del 0, 2 líneas del 2, etc.)?

Las operaciones estándar de E/S en C no proporcionan soluciones simples a dicho problema; así la E/S sigue siendo un objeto de investigación por parte de la comunidad del procesamiento paralelo.

Por todo ello asumiremos que el proceso 0 puede al menos escribir en la salida estándar. También asumiremos que puede leer de la entrada estándar. En todos los casos asumiremos que sólo el proceso 0 puede interactuar con la E/S. Esto podría parecer una debilidad, ya que como mencionamos antes las máquinas paralelas permiten que múltiples procesos interactúen con la E/S. Sin embargo es un método muy sólido que nos permitirá realizar un manejo limpio de la E/S en nuestros programas [4].

2.2.4 UBICACION DE LOS PROCESOS

La función **MPI_Get_processor_name()** nos permite conocer el nombre del procesador donde está ubicado cada proceso. Esto puede ser útil para monitorizar nuestros programas en redes heterogéneas. Conocer en qué máquina concreta se está ejecutando un proceso específico puede ser determinante para explicar su comportamiento, para lo cual podemos ayudarnos con las herramientas de monitorización.
La sintaxis de dicha función es la siguiente:

int MPI_Get_processor_name(char* nombre, int* longnombre)

El parámetro **nombre** es una cadena (vector de caracteres) cuyo tamaño debe ser al menos igual a la constante **MPI_MAX_PROCESSOR_NAME**. En dicho vector quedará almacenado el nombre del procesador. EL parámetro **longnombre** es otro parámetro de salida que nos informa de la longitud de la cadena obtenida.

2.2.5 INFORMACIÓN TEMPORAL

Con el objeto de facilitar el control y la monitorización, **MPI** nos ofrece funciones encaminadas a proporcionar información temporal sobre la ejecución de nuestros programas. La función **MPI_Wtime()** nos informa sobre el tiempo transcurrido en un determinado proceso.
Dicha función no tiene parámetros; su valor de retorno es de tipo **double** y representa el tiempo transcurrido en segundos desde el comienzo de la ejecución del proceso. Si el atributo MPI_WTIME_IS_GLOBAL está definido y su valor es **true** (por defecto es así) el valor devuelto por la función estará sincronizado con todos los procesos pertenecientes al comunicador MPI_COMM_WORLD.

Utilizaremos esta función para ofrecer cierta información al usuario sobre el tiempo de ejecución de nuestro programa por la salida estándar. Nos permitirá en estos casos diferenciar entre el tiempo de ejecución de un programa y el tiempo de procesamiento.

Donde:

Tiempo de ejecución de un programa es el tiempo que emplea para resolver un problema en un computador paralelo, desde el momento en que es generado el primer proceso hasta que acaba el último. Este tiempo no debe ser calculado con la función **MPI_Wtime( )** debido a que dicha función forma parte del entorno MPI, el cual debe ser finalizado antes de que el programa termine. Por lo tanto el tiempo que arrojaría es incorrecto, siempre menor al que debería; en su lugar se debe usar las herramientas de monitorización para determinar el tiempo de ejecución.

Por otro lado llamamos tiempo de procesamiento al tiempo que emplea el computador en calcular el resultado, eliminando el tiempo de inicialización y finalización de los procesos, la comunicación con el usuario. De esta manera si el usuario tarda diez

segundos en introducir los datos para un problema, ese tiempo no será añadido al tiempo de procesamiento pero sí al tiempo de ejecución. La función **MPI_Wtime()** nos ayudará a calcular el tiempo de procesamiento de un programa para después mostrarlo al usuario por la salida estándar.

En el caso del algoritmo ¡Hola Mundo! el tiempo de procesamiento no es indicativo. Sin embargo es muy útil en programas donde un proceso (normalmente el proceso 0) desencadena la ejecución de los demás (normalmente al distribuir entre todos los procesos los dato que el usuario introduce por la entrada estándar). Este será el caso más común entre los algoritmos expuestos en el presente documento.

2.2.6 IMPLEMENTACIÓN ALGORITMO ¡Hola Mundo!

A continuación exponemos el código del algoritmo ¡Hola Mundo!. Como vemos, dicho programa utiliza el paradigma SPMD ("Same Program Multiple Data", Programa único y flujo de datos Múltiple). Obtenemos el efecto de diferentes programas ejecutándose en distintos procesadores bifurcando la ejecución del programa. Para ello nos basamos en el identificador de otros procesos aunque todos los procesos estén ejecutando el mismo programa. Este es el método más común para escribir programas MIMD.

Algoritmo 4.1 ¡Hola Mundo!

```
#define TAMCADENA 100
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char **argv)
{
  int id;                          /*Identificador del proceso*/
  int numprocs;                    /*Número de procesos*/
  char nombreproc[MPI_MAX_PROCESSOR_NAME];/*Nombre del procesador*/
  int long_nombreproc;                  /*Longitud nombre procesa*/
  double t_inic=0.0                /*Tiempo inicio de la ejecución*/
  double t_final;                  /*Tiempo final de la ejecución*/
```

```
 /* Inicia el entorno MPI */
 MPI_Init(&argc, &argv);

 /*Almacenamos el identificador del proceso*/
  MPI_Comm_rank(MPI_COMM_WORLD, &id);

/*Almacenamos el número de procesos*/
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  /*E/S:nombre del procesador, proceso 0*/
  MPI_Get_processor_name(nombreproc,&long_nombreproc);
  if(id==0){
        printf("\nProceso   %d   en   %   Encargado   de   la   E/S
\n\n",id,nombreproc);
  }
 /*Tiempo inicial de la ejecución*/
  t_inic=MPI_Wtime();
 if (id==0){
      printf("Hola, Mundo!\n");
      }
/*Tiempo Final de la ejecución*/
  t_final=MPI_Wtime();

  /*E/S: Información sobre la ejecución*/
  if(id==0){
      printf("Numero de Procesos: %d\n",numprocs);
      printf("Tiempo de Procesamiento: %f\n\n", t_final - t_inic);
  }
  /*Finalizamos el entorno de ejecución de MPI*/
  MPI_Finalize();
  return 0;
}
```

CAPITULO 3
EJEMPLOS DE APLICACIONES USANDO MPI

INTRODUCCIÓN

MPI está especialmente diseñado para desarrollar aplicaciones SPMD. Al arrancar una aplicación se lanzan en paralelo N copias del mismo programa (procesos). Estos procesos no avanzan sincronizados instrucción a instrucción sino que la sincronización, cuando sea necesaria, tiene que ser explícita. Los procesos tienen un espacio de memoria completamente separado. El intercambio de información, así como la sincronización, se hacen mediante paso de mensajes.

Se dispone de funciones de comunicación punto a punto (que involucran sólo a dos procesos), y de funciones u operaciones colectivas (que involucran a múltiples procesos). Los procesos pueden agruparse y formar *comunicadores*, lo que permite una definición del ámbito de las operaciones colectivas, así como un diseño modular. En este capítulo se ejemplifican estos tipos de comunicación, mediante la implementación de 3 algoritmos, a saber, el algoritmo de Montecarlo, la regla del trapecio y por último multiplicación de matrices de fox.

3.1 PASO DE MENSAJES

El paso de mensajes es quizá la función más importante y característica del estándar MPI. Se utiliza básicamente para el intercambio de datos entre los procesos en ejecución. En este capítulo estudiaremos las características más importantes de los mensajes *bloqueantes* y *no bloqueantes*, aplicando dicha teoría a la implementación de un algoritmo diseñado para el cálculo de áreas circulares.

3.1.1 ALGORITMO CALCULO DE AREAS MEDIANTE MONTECARLO

El método de Montecarlo es un método numérico que permite resolver problemas matemáticos mediante la simulación de variables aleatorias. Utilizaremos dicho método para aproximar el área de un círculo con radio determinado.
Supongamos que queremos calcular el área de un círculo de radio *r*. Para hacerlo tomaremos un cuadrado de lado ***2r*** y lo pondremos alrededor del círculo de radio ***r***, de manera que quede circunscrito. A continuación generaremos un número de puntos aleatorios dentro del área del cuadrado. Algunos caerán dentro del área del círculo y otros no, como se expone en la figura 3.1.

Tomaremos ***N*** como el número de puntos aleatorios generados dentro del cuadrado y ***n*** como el número de puntos que caen dentro del círculo. Si sabemos que el área del círculo es ***p*** y que el área del cuadrado es (***2r***), tendremos la siguiente igualdad:

$$\pi r^2 = \frac{n}{N}(2r)^2$$

Con lo cual llegamos fácilmente a que:

$$\pi = 4 \times \frac{n}{N}$$

Utilizando dicha aproximación de **Π** y sabiendo el radio del círculo aproximaremos fácilmente su área mediante **Πr²**. Lógicamente cuantas más muestras tomemos, es decir, cuantos más puntos generemos, mejor será la aproximación de **Π**. También interviene mucho la aleatoriedad de las muestras; cuanto más aleatorias sean, mejor.
La cuestión ahora es determinar los puntos que caen dentro del círculo y los que caen fuera. Para explicarlo observemos la figura 3.2. Dado un punto ***(x, y)*** podremos saber si ha caído dentro del círculo mediante la regla de Pitágoras:

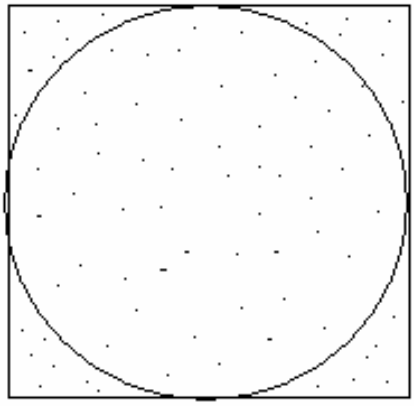

Fig. 3.1Generación de Puntos aleatorios

$$h^2 = x^2 + y^2$$

Si la hipotenusa es mayor que el radio del círculo querrá decir que el punto está afuera de dicho círculo. Si la hipotenusa es menor que el radio, consideraremos que está adentro. Como podemos observar en la figura, también podemos generar puntos en sólo una cuarta parte del cuadrado y el círculo, de manera que la proporción no queda alterada. Así lo hacemos en la implementación del algoritmo Cálculo de Áreas mediante Montecarlo.

3.1.2 EL ENTORNO DEL MENSAJE

El paso de mensajes bloqueantes se lleva a cabo en nuestros programas por las funciones **MPI_Send()** y **MPI_Recv()** principalmente. La primera función envía un mensaje a un proceso determinado. La segunda recibe un mensaje de un proceso. Éstas son las funciones más básicas de paso de mensajes en **MPI**. Para que el mensaje sea comunicado con éxito, el sistema debe adjuntar alguna información a los datos que el programa intenta transmitir. Esta información adicional conforma el *entorno* del mensaje. En **MPI** el entorno contiene la siguiente información:

- El identificador del proceso receptor del mensaje.
- El identificador del proceso emisor del mensaje.
- Una etiqueta
- Un comunicador

Estos datos pueden ser usados por el proceso receptor para distinguir entre los mensajes entrantes. El origen puede ser usado para distinguir mensajes recibidos por distintos procesos.

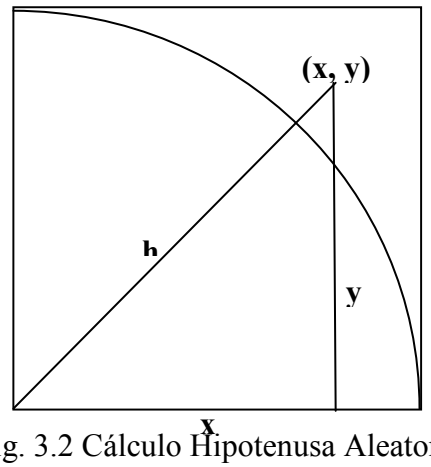

Fig. 3.2 Cálculo Hipotenusa Aleatorio

La *etiqueta* es un **int** especificado por el usuario que puede ser usado para distinguir mensajes recibidos por un único proceso. Por ejemplo, supongamos que el proceso A está enviando dos mensajes al proceso B; ambos mensajes contienen un número flotante. Uno de los flotantes se emplea en un cálculo, mientras que otro debe ser impreso. Para determinar cuál es cuál, A puede usar etiquetas diferentes para cada mensaje. Si B usa las mismas etiquetas que A en las recepciones correspondientes, "sabrá" qué hacer con ellas. MPI garantiza que los enteros dentro del intervalo 0-32767 pueden ser usados como etiquetas. La mayoría de las implementaciones permiten valores mucho mayores.
Como dijimos antes un *comunicador* es básicamente una colección de procesos que pueden mandarse mensajes entre sí. La importancia de los comunicadores se acentúa cuando los módulos de un programa han sido escritos independientemente de los demás. Por ejemplo, supongamos que queremos resolver un sistema de ecuaciones diferenciales y, en el transcurso de resolverlo, tenemos que resolver un sistema de ecuaciones lineales. Mejor que escribir la resolución del sistema de ecuaciones lineales desde el principio, podríamos usar una librería de funciones escrita por otra persona y optimizada para el sistema que estamos usando. ¿Cómo evitamos que se confundan los mensajes que nosotros enviamos entre los procesos A y B con los mensajes enviados en la librería de funciones? Sin la ventaja de los comunicadores probablemente haríamos una partición del rango de las posibles etiquetas, haciendo que parte de ellas sólo puedan ser usadas por la librería de funciones. Esto es tedioso y puede causarnos problemas si ejecutamos el programa en otro sistema: puede que la librería del otro sistema o haga uso del mismo

rango de etiquetas. Con la ventaja de los comunicadores, simplemente creamos un comunicador para uso exclusivo en la resolución del sistema de ecuaciones lineales, y se lo pasamos a la librería encargada de hacerlo como argumento en la llamada [4][5][6].

Comentaremos los detalles más adelante. Por ahora continuaremos utilizando el comunicador **MPI_COMM_WORLD**, el cual consiste en todos los procesos que se ejecutan cuando el programa comienza.

Tipos de Datos MPI	Tipos de Datos 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_BYTE	long double
MPI_PACKED	

Cuadro 3.1: Tipos de Datos MPI

3.1.3 FUNCIONES DE PASO DE MENSAJES BLOQUEANTES

Para resumir detallamos la sintaxis de las dos funciones más importantes de paso de mensajes bloqueantes:

```
int MPI_Send(void* mensaje,int contador,MPI_Datatype tipo_datos,
                int destino, int etiqueta, MPI_Comm comunicador)

int MPI_Recv(void* mensaje, int contador, MPI_Datatype
     tipo_datos, int origen, int etiqueta, MPI_Comm comunicador,
     MPI_Status* status)
```

Al igual que la mayoría de las funciones de la biblioteca estándar de C, la mayoría de las funciones MPI retornan un entero como código de error. Sin embargo, como la ma-

yoría de los programadores de C, ignoraremos esos valores de retorno en casi todos los casos.

Los contenidos del mensaje son almacenados en un bloque de memoria referenciado por el argumento **mensaje**. Los siguientes dos argumentos, **contador** y **tipo_datos**, permiten al sistema identificar el final del mensaje: éste contiene una secuencia de **contador** valores, cada uno del tipo MPI **datatype**. Este tipo no es un tipo de C, aunque la mayoría de los tipos predefinidos corresponden a tipos de C. En el cuadro 2.1 se listan los tipos predefinidos de MPI con sus correspondientes tipos de C, si éstos existen.

Los últimos dos tipos, **MPI_BYTE** y **MPI_PACKED**, no corresponden con tipos estándar de C. El tipo **MPI_BYTE** puede ser usado si queremos forzar que el sistema no realice conversión alguna entre las distintas representaciones de los datos (por ejemplo, en una red heterogénea de estaciones de trabajo que utilicen una representación de datos distinta).

Notar que la cantidad de espacio reservado para el buffer de entrada no tiene porqué coincidir con la cantidad exacta de espacio que ocupa el mensaje que estemos recibiendo. Por ejemplo, podría darse el caso de que el tamaño del mensaje que el proceso 1 envía al proceso 0 sea de 28 caracteres (**strlen(mensaje)+1**), aunque el proceso 0 reciba el mensaje en un buffer que tiene capacidad para 100 caracteres. Esto tiene sentido. En general el proceso receptor no conoce el tamaño exacto del mensaje que se le está enviando. MPI permite recibir mensajes tan largos como capacidad reservada tengamos. Si no tenemos suficiente capacidad, tendremos un error de desbordamiento.

Los argumentos **destino** y **origen** son los identificadores del proceso receptor y del proceso emisor, respectivamente. MPI permite que el argumento **origen** sea un *comodín*. Existe una constante predefinida llamada **MPI_ANY_SOURCE** que puede ser usada si un proceso está preparado para recibir un mensaje procedente de cualquier proceso. No existe comodín para **destino**.

Como dijimos anteriormente MPI contiene dos mecanismos diseñados específicamente para "particionar el espacio de los mensajes": las etiquetas y los comunicadores. Los argumentos **etiqueta** y **comunicador** son, respectivamente, la etiqueta y el comunicador. El argumento **etiqueta** es un **int** y, por ahora, nuestro único comunicador es

MPI_COMM_WORLD, el cual, como comentamos antes, está predefinido en todos los sistemas MPI y consiste en todos los procesos que se ejecutan cuando el programa comienza. Existe un comodín, **MPI_ANY_TAG**, que puede ser usado en **MPI_Recv()** para la etiqueta. No existe un comodín para el comunicador. En otras palabras, para que el proceso *A* mande un mensaje al proceso *B* el argumento **comunicador** que *A* usa en **MPI_Send()** debe ser idéntico al argumento que *B* usa en **MPI_Recv()**.

El último argumento de **MPI_Recv()**, **status**, retorna información acerca de los datos que hemos recibido. Referencia a un registro con dos campos, uno para el origen y otro para la etiqueta. Por ejemplo, si el origen era **MPI_ANY_SOURCE** en la llamada a **MPI_Recv()**, el argumento **status** contendrá el identificador del proceso que envió el mensaje[5][6].

3.1.4 FUNCIONES DE PASO DE MENSAJES NO BLOQUEANTES

El rendimiento de los sistemas de paso de mensajes puede ser mejorado mediante el solapamiento entre comunicación y procesamiento en los algoritmos. Un método para realizar esto es mediante la comunicación no bloqueante. Las funciones de envío de mensajes no bloqueantes inician el envío del mensaje, pero no lo completan; para ello se necesita una llamada a la función de finalización de envíos no bloqueantes. De la misma manera una llamada a una función de recepción de mensajes no bloqueantes no detiene la ejecución del algoritmo hasta que dicho mensaje sea realmente recibido, como ocurre con las versiones bloqueantes. De este modo podemos solapar comunicación y procesamiento en nuestros algoritmos.

Las versiones no bloqueantes de las funciones de paso de mensajes son:

```
int MPI_Isend(void* mensaje, int contador, MPI_Datatype
     tipo_datos, int destino, int etiqueta, MPI_Comm comunica-
dor, MPI_Request* peticion)

int MPI_Irecv(void* mensaje, int contador, MPI_Datatype
     tipo_datos, int origen, int etiqueta, MPI_Comm comunicador,
     MPI_Request* peticion)
```

A simple vista la diferencia entre estas funciones y sus versiones bloqueantes es la existencia del argumento **peticion** en sus llamadas. Este argumento es utilizado por las funciones de finalización de transmisiones bloqueantes para su ejecución.

Las funciones de finalización de transmisiones bloqueantes son:

```
int MPI_Wait(MPI_Request* peticion, MPI_Status* status)
int MPI_Waitany(int contador, MPI_Request* vector_peticiones,
                int* indice, MPI_Status* status)
int MPI_Waitall(int contador, MPI_Request* vector_peticiones,
                MPI_Status* status)
```

La función **MPI_Wait()** espera a que se complete un envío o una recepción de un mensaje. Para ello le pasamos como argumento de entrada el argumento de salida que nos proporciona la función de paso de mensajes no bloqueantes en el momento de la llamada. El argumento de salida **status** nos proporciona información acerca de los datos que hemos recibido.

Por su parte **MPI_Waitany()** espera a que se complete cualquiera de las peticiones que le pasamos en **vector_peticiones**, devolviendo en el argumento **indice** la posición de la petición satisfecha. **MPI_Waitall()** espera a que se completen todas las peticiones que le pasamos en **vector_peticiones**.

3.1.5 AGRUPACIONES DE DATOS

Con la generación de máquinas actual el envío de mensajes es una operación costosa. Según esta regla, cuanto menos mensajes enviemos mejor será el rendimiento general del algoritmo. Así pues en el algoritmo Cálculo de Áreas mediante Montecarlo podríamos mejorar el rendimiento si distribuimos los datos de entrada en un solo mensaje entre los procesadores.

Recordemos que las funciones de paso de mensajes tienen dos argumentos llamados **contador** y **tipo_datos**. Estos dos parámetros permiten al usuario agrupar datos que tengan el mismo tipo básico en un único mensaje. Para usarlo dichos datos deben estar almacenados en direcciones de memoria contiguas. Debido a que el lenguaje C garanti-

za que los elementos de un vector están almacenados en direcciones de memoria contiguas, si queremos enviar los elementos de un vector, o un subconjunto de ellos, podremos hacerlo en un solo mensaje.

Desafortunadamente esto no nos ayuda en el algoritmo Cálculo de Áreas mediante Montecarlo si queremos enviar los datos de entrada a cada proceso en un solo mensaje. Ello es debido a que las variables que contienen el radio y el número de muestras no tienen porqué estar alojadas en direcciones de memoria contiguas. Tampoco debemos almacenar estos datos en un vector debido a que ello restaría claridad y elegancia al código. Para lograr nuestro objetivo utilizaremos una facilidad de MPI para la agrupación de los datos.

3.1.6. TIPOS DERIVADOS

Una opción razonable para conseguir nuestro objetivo podría ser almacenar el radio y el número de muestras en una estructura con dos miembros (un flotante largo y un entero largo) como se muestra a continuación:

```
typedef struct{
      long double radio;
      long nmuestras_local;
} Tipo_Datos_Entrada;
```

luego intentar utilizar el tipo de datos de la estructura como argumento **tipo_datos** en las funciones de paso de mensajes. El problema aquí es que el tipo de datos de la estructura no es un tipo de datos MPI; necesitamos construir un tipo de datos MPI a partir del tipo de datos de C. MPI propone una solución a esto permitiendo al usuario construir tipos de datos MPI en tiempo de ejecución. Para construir un tipo de datos MPI básicamente se especifica la distribución de los datos en el tipo (los tipos de los miembros y sus direcciones relativas de memoria). Un tipo de datos MPI construido de este modo se denomina *tipo de datos derivado*. Para ver cómo funciona esto expondremos la función que construye dicho tipo de datos.

```
void construir_tipo_derivado(Tipo_Datos_Entrada* pdatos, MPI_Datatype*
pMPI_Tipo_Datos){
   1.       MPI_Datatype tipos[2];
   2.       int longitudes[2];
   3.       MPI_Aint direcc[3];
   4.       MPI_Aint desplaz[2];
```

```
5.        tipos[0]=MPI_LONG_DOUBLE;
6.        tipos[1]=MPI_LONG;
7.        longitudes[0]=1;
8.        longitudes[1]=1;

9.        MPI_Address(pdatos,&direcc[0]);
10.       MPI_Address(&(pdatos->radio),&direcc[1]);
11.       MPI_Address(&(pdatos->nmuestras_local),&direcc[2]);
12.       desplaz[0]=direcc[1]-direcc[0];
13.       desplaz[1]=direcc[2]-direcc[0];

14. MPI_Type_struct(2,longitudes,desplaz,tipos,pMPI_Tipo_Datos;
15.       MPI_Type_commit(pMPI_Tipo_Datos);
16.       }
```

Líneas 1-2: Especifican los tipos de los miembros del tipo de datos derivado.

Líneas 3-4: Especifican el número de elementos de cada tipo.

Líneas 9-13: La función **MPI_Address()** nos ayuda a calcular los desplazamientos de cada uno de los miembros con respecto a la dirección inicial del primero. Con esta información ya sabemos los tipos, los tamaños y las direcciones relativas de memoria de cada uno de los miembros del tipo de datos de C.

Línea 14: Definimos el tipo de datos MPI derivado del tipo de datos de C. Lo logramos al llamar a las funciones **MPI_Type_struct()** y **MPI_Type_Commit()**.

Linea 15: Certificarlo de manera que pueda ser usado

El tipo de datos MPI creado puede ser usado en cualquiera de las funciones de comunicación de MPI. Para usarlo simplemente usaremos como primer argumento de las funciones la dirección inicial de la estructura a enviar, y el tipo de datos MPI creado en el argumento **tipo_datos**.

3.1.7. VECTORES

La función **MPI_Type_vector()** crea un tipo derivado consistente en **contador** elementos.

```
int MPI_Type_vector(int contador, int longitud_bloque,
                int salto, MPI_Datatype tipo_datos_elem,
                MPI_Datatype* nuevo_tipo)
```

Cada elemento contiene **longitud_bloque** elementos de tipo **tipo_datos_elem**. El argumento **salto** es el número de elementos de tipo **tipo_datos_elem** que hay entre los sucesivos elementos de **nuevo_tipo**.

3.1.8 NOTAS IMPORTANTES DE LA IMPLEMENTACIÓN CÁLCULO DE ÁREAS MEDIANTE MONTECARLO

A continuación exponemos las versiones bloqueante y no bloqueante del algoritmo Cálculo de Áreas mediante Montecarlo.

IMPLEMENTACIÓN CON MENSAJES BLOQUEANTES

```
1.  int main(int argc, char** argv){
2.  ...
3.  MPI_Status status;
4.  int origen;
5.  MPI_Init(&argc,&argv);
6.  MPI_Comm_rank(MPI_COMM_WORLD,&id);
7.  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
8.  MPI_Get_processor_name(nombreproc,&lnombreproc);
9.  if (id == 0)
10. {printf("\nProceso %d en %s Encargado de
    E/S\n",id,nombreproc);}
11. if (id==0){
12. obtener_datos(&radio,&nmuestras_global);
13. nmuestras_local=nmuestras_global/numprocs;
14. nmuestras_global=nmuestras_local*numprocs;}
15. if(id==0){printf("Procesando...\n\n");}
16.  tinic = MPI_Wtime();
17. ...
```

Líneas 1-4: Declaración de variables.

Línea 5: Inicializamos MPI.

Línea 6: Almacenamos el identificador del proceso.

Línea 7: Almacenamos el número de procesos.

Línea 8: Nombre del procesador, procesador 0.

Líneas 9 y 10: Si es el maestro distribuye los intervalos.

Línea 11 y 12: Obtenemos datos introducidos por el usuario.
Línea 13: Truncamos número de muestras global.

Línea 14: Aproximación general (media de todas las locales).

Línea 15 y 16: Tiempo Inicial de procesamiento

```
int distribuir_datos(int id, int numprocs, long double* pradio, long*
pnmuestras_local){
19. MPI_Datatype MPI_Tipo_Datos_Entrada;
20. Tipo_Datos_Entrada datos_entrada;
21. int origen=0;
22. int destino;
23. int etiqueta;
24. MPI_Status status;
  ...
25. if(id==0){
26. datos_entrada.radio=*pradio;
27. datos_entrada.nmuestras_local=*pnmuestras_local;
28. for(destino=1;destino<numprocs;destino++){
29. etiqueta=30;
30. MPI_Send(&datos_entrada,1,MPI_Tipo_Datos_Entrada,destino,etiqueta,MPI_COMM_WOR
    LD);}
31. }  //del if
```

```
   32. else{/*recepción de los datos*/
          etiqueta=30;
   33. MPI_Recv(&datos_entrada,1,MPI_Tipo_Datos_Entrada,origen,etiqueta,MPI_COMM_WORL
       D,&status);
   34. *pradio=datos_entrada.radio;
   35. *pnmuestras_local=datos_entrada.nmuestras_local;
   36. }
   37. return 0;
}
```

Línea 21: Proceso origen de mensajes

Línea 22: Proceso destino de datos

Línea 26 y 27: Asignación de los datos introducidos por el usuario a la estructura.

Línea 30: Se envían los datos a los diferentes procesadores con ayuda de la función MPI_Send(), donde el mensaje está contenido en la estructura *datos_entrada*, y el tipo de datos es el tipo derivado construido MPI_Tipo_Datos_Entrada.

```
...
long double MonteCarlo(int id, long double radio, long nummuestras){
...
   19. srand(id);
   20. for(i=0;i!=nummuestras;i++){
   21. x=(long double)rand()/RAND_MAX;
   22. y=(long double)rand()/RAND_MAX;
   23. hip=sqrt(pow(x,2)+pow(y,2));
   24. if(hip>1){fallos++;}
   25. else{aciertos++;}
   26. }
   27. pi=4.00*((long double)aciertos/nummuestras);
   28. ...
}
```

Línea 19: Utilizamos como semilla el identificador del proceso.

Líneas 20-26: Estimación de pi

Línea 27: Valor aproximado de pi

3.1.10 IMPLEMENTACION CON MENSAJES NO BLOQUEANTES

```
int distribuir_datos(int id, int numprocs, long double* pradio, long*
pnmuestras_local){
       ...
   19. if(id==0){ /*envio de los datos*/
   20.  datos_entrada.radio=*pradio;
   21.  datos_entrada.nmuestras_local=*pnmuestras_local;
   22.  for(destino=1;destino<numprocs;destino++){
   23.  etiqueta=30;
   24. MPI_Isend(&datos_entrada,1,MPI_Tipo_Datos_Entrada,destino,etiqueta,MPI_COMM
       _WORLD,&request);}
   25. }
   26.  else{
   27.  etiqueta=30;
       MPI_Irecv(&datos_entrada,1,MPI_Tipo_Datos_Entrada,origen,etiqueta,MPI_COMM_
       WORLD,&request);
   28. MPI_Wait(&request,&status);
   29. *pradio=datos_entrada.radio;
   30. *pnmuestras_local=datos_entrada.nmuestras_local;
   31. }
   32. return 0;
}
```

Los pasos a seguir son los mismos a excepción de la línea 24, que se describe a continuación:
Línea 24: Se envían los datos ahora con paso de mensajes no bloqueante con ayuda de la función MPI_Isend().

3.2. COMUNICACIÓN COLECTIVA

Son aquellas que se aplican al mismo tiempo a todos los procesos pertenecientes a un comunicador. Tienen una gran importancia en el estándar MPI, debido a la claridad de su sintaxis y a su eficiencia. Para analizar su utilidad y conveniencia implementaremos el algoritmo Regla del Trapecio de manera que haga un uso inteligente de dichas operaciones.

3.2.1 REGLA DEL TRAPECIO

La regla del trapecio estima $\int_a^b f(x)dx$ mediante la división del intervalo [a, b] en *n* segmentos iguales, calculando la siguiente suma:

$$h\left[f(x_0)/2 + f(x_n)/2 + \sum_{i=1}^{n-1} f(x_i) \right]$$

Donde $h = (b-a)/n$, y $x_i = a + ih$, $i = 0, \ldots, n$.

Una manera de realizar este cálculo mediante un algoritmo paralelo es dividir el intervalo [a, b] entre los procesos haciendo que cada proceso calcule la integral de ***f*(x)** sobre su subintervalo. Para estimar la integral completa se suman los cálculos locales de cada proceso. Supongamos que tenemos *p* procesos y *n* segmentos, y, para simplificar, supongamos que ***n*** es divisible entre ***p.*** En este caso es natural que el primer proceso calcule la integral de los primeros ***n/p*** segmentos, el segundo proceso calculará el área de los siguientes ***n/p*** segmentos, etc. Siguiendo esta regla el proceso *q* estimará la integral del subintervalo:

$$\left[a + q\frac{nh}{p}, a + (q+1)\frac{nh}{p} \right]$$

Cada proceso necesita la siguiente información:

- El número de procesos ***p***
- Su identificador
- El intervalo de integración completo ***[a , b]***
- El número de segmentos **n**
- La función f(x)a integrar

Los dos primeros datos se pueden conseguir llamando a las funciones **MPI_Comm_size()** y **MPI_Comm_rank()**. Sin embargo tanto el intervalo **[a, b]** como el número de segmentos ***n*** deben ser introducidos por el usuario. Por otro lado sería positivo que el usuario pudiera elegir la función f(x) a integrar (dentro de un conjunto predefinido de funciones). Recordemos que sólo un proceso (normalmente el proceso 0) debe encargarse de obtener los datos de la entrada estándar y distribuirlos entre los demás procesos. Esto puede hacerse mediante paso de mensajes o mediante comunicación colectiva, siendo la segunda opción más eficiente.

Por otro lado, una buena forma de sumar los cálculos individuales de los procesos es hacer que cada proceso envíe su cálculo *local* al proceso 0, y que el proceso 0 haga la suma final. De nuevo podemos emplear al menos dos técnicas: paso de mensajes y operaciones de reducción. Dado que las operaciones de reducción son más eficientes, utilizaremos la segunda opción.

Las funciones que ofreceremos al usuario estarán codificadas en funciones de código C, por lo que tendremos que modificar el código y compilarlo de nuevo para introducir nuevas funciones. Entre las funciones ofrecidas incluiremos al menos las siguientes:

- $f(x) = 2x$
- $f(x) = x^2$
- $f(x) = \frac{1}{x}$
- $f(x) = \frac{4}{(1+x^2)}$

Donde la última función nos permite realizar la búsqueda del número **Π**. El área bajo la curva $y = 1/(1+x^2)$ nos proporciona un método para computar **Π:**

$$\int_0^1 \frac{4}{(1+x^2)} dx = [4\arctan(x)]_0^1 = 4\left(\frac{\pi}{4} - 0\right) = \pi$$

Por lo tanto tenemos que para computar **Π** mediante la regla del trapecio utilizaremos, $f(x) = \frac{4}{(1+x^2)}$ $a = 0$ y $b = 1$.

3.2.2 DISTRIBUCIÓN Y RECOLECCIÓN DE LOS DATOS

La distribución y recolección de los datos se puede realizar mediante dos técnicas*: paso de mensajes* (como hicimos en el algoritmo Cálculo de Áreas mediante Montecarlo) y *comunicación colectiva*. Las operaciones de comunicación colectiva son más eficientes debido a que reducen el tiempo empleado en dichas operaciones, reduciendo el número de pasos necesarios para su ejecución.
En la técnica de paso de mensajes el problema radica en que cargamos a un proceso con demasiado trabajo. Al utilizar ésta técnica para distribuir los datos damos básicamente los siguientes pasos:

1. Proceso 0 recoge los datos de la entrada estándar
2. Proceso 0 envía un mensaje a cada uno de los restantes procesos comunicándole los datos de entrada.

Si utilizamos un bucle para realizar el segundo paso, los procesos con identificador mayor deben continuar esperando mientras el proceso 0 envía los datos de entrada a los procesos con identificador menor. Éste no es sólo un asunto de E/S y por supuesto no es un comportamiento deseable: el punto principal del procesamiento paralelo es conseguir que múltiples procesos colaboren para resolver un problema. Si uno de los procesos está haciendo la mayoría del trabajo, podríamos usar mejor una máquina convencional de un solo procesador. Por otro lado observamos que existe una ineficiencia similar en el final del algoritmo al utilizar paso de mensajes en el proceso de recolección, donde el proceso 0 hace todo el trabajo de recolectar y sumar las integrales locales.
La pregunta que surge es... ¿cómo podemos dividir el trabajo más equitativamente entre los procesos? Una solución natural es imaginar que tenemos un árbol de procesos, con el proceso 0 como raíz.
Durante la primera etapa de la distribución de los datos pongamos que el proceso 0 envía los datos al 4. Durante la siguiente etapa el 0 envía los datos al 2, mientras que el 4 los envía al 6. Durante la última etapa el 0 se los envía al 1, mientras el 2 los envía al 3, el 4 al 5 y el 6 al 7 (figura 3.2.1). Así reducimos el bucle de distribución de la entrada de 7 etapas a 3 etapas. De manera más general, si tenemos ***p*** procesos este procedimiento

permite distribuir los datos de entrada en $\log_2(p)$ etapas, mientras que de la anterior manera lo hacíamos en $(p-1)$ etapas. Si ***p*** es grande esta mejora es una gran ventaja.

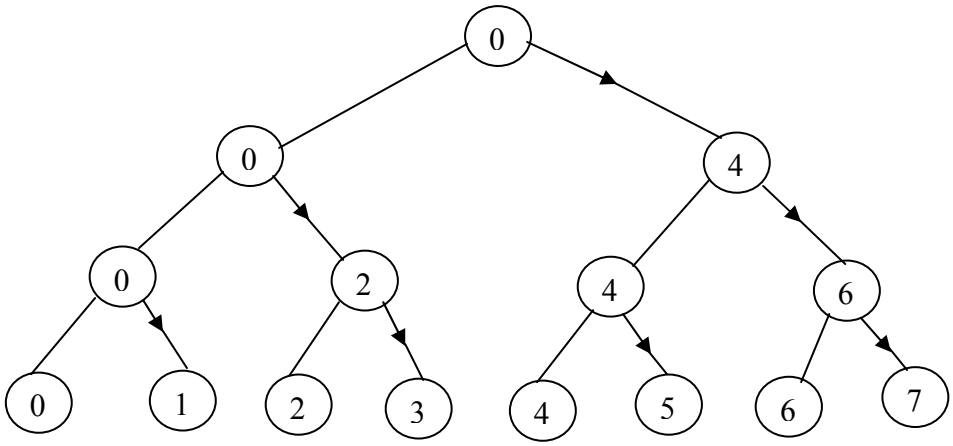

Fig. 3.2.1: Procesos estructurados en árbol

Si queremos utilizar un esquema de distribución estructurado en árbol en nuestro algoritmo, necesitamos introducir un bucle con $\log_2$ ***(p)*** etapas. En la implementación de dicho bucle cada proceso necesita calcular en cada etapa:

- Si recibe, y en caso de hacerlo, el origen: y
- Si envía, y en caso de hacerlo, el destino

Estos cálculos pueden ser algo complicados, especialmente porque no hay un ordenamiento fijo. En nuestro ejemplo elegimos:

1. 0 envía al 4
2. 0 envía al 2, 4 al 6
3. 0 envía al 1 , 2 al 3, 4 al 5, 6 al 7

Aunque podríamos haber elegido también:

1. 0 envía al 1
2. 0 envía al 2, 1 al 3
3. 0 envía al 4, 1 al 5, 2 al 6, 3 al 7

De hecho, a menos que sepamos algo acerca de la topología subyacente de nuestra máquina no podremos decidir realmente qué esquema es mejor.
Lo ideal sería usar una función diseñada específicamente para la máquina que estemos usando de manera que no nos tengamos que preocupar acerca de todos los detalles, y que no tengamos que modificar el código cada vez que cambiamos de máquina. Y como habrá podido adivinar MPI proporciona dicha función.

3.2.3 OPERACIONES DE COMUNICACIÓN COLECTIVA

Un patrón de comunicación que engloba a todos los procesos de un comunicador es una *comunicación colectiva*. Como consecuencia una comunicación colectiva implica a más de dos procesas. MPI incluye una serie de operaciones colectivas:

- Barreras de sincronización
- Broadcast (difusión)
- Gather (recolección)
- Scatter (distribución)
- Operaciones de reducción (suma, multiplicación, mínimo, etc.)

3.2.4 BARRERAS Y BROADCAST

Dos de las operaciones colectivas mas comunes son las barreras de sincronización (MPI_Barrier()) y el envío de información en modo difusión (MPI_Broadcast()).
La Barrera o *Barrier* no exige ninguna clase de intercambio de información. Es una operación de sincronización, que bloquea a los procesos de un comunicador hasta que todos ellos han pasado por la barrera. Suele emplearse para dar por finalizada una etapa de programa, asegurándose de que todos han terminado antes de dar comienzo a la siguiente. En MPI la función para hacer esto es:

```
int MPI_Barrier(MPI_Comm com);
```

Un *broadcast* es una operación de comunicación colectiva en donde un proceso envía un mismo mensaje a todos los procesos. En MPI la función para hacer esto es **MPI_Bcast()**:

```
int MPI_Bcast(void* mensaje, int contador,
          MPI_Datatype tipo_datos, int raiz,MPI_Comm com);
```

La función envía una copia de los datos contenidos en **mensaje** por el proceso **raíz** a cada proceso perteneciente al comunicador **com**. Debe ser llamada por todos los procesos pertenecientes al comunicador con los mismos argumentos para **raiz** y **com**. Por lo tanto un mensaje broadcast no puede ser recibido con la función **MPI_Recv()**. Los parámetros **contador** y **tipo_datos** tienen la misma finalidad que en **MPI_Send()** y **MPI_Recv()**: especifican la extensión del mensaje. Sin embargo, al contrario que en las funciones punto a punto, MPI insiste en que en las operaciones de comunicación colectiva **contador** y **tipo_datos** deben ser iguales en todos los procesos pertenecientes al comunicador. La razón de esto es que en algunas operaciones colectivas un único proceso recibe datos de otros muchos procesos, y para que el programa determine qué cantidad de datos han sido recibidos necesitaría un vector de status de retorno [3][6].

3.2.5 GATHER (RECOLECCIÓN)

La función Gather realiza una colección de datos en el proceso raíz. Este proceso recopila un vector de datos, al que contribuyen todos los procesos del comunicador con la misma cantidad de datos. El proceso raíz almacena las contribuciones de forma consecutiva.

La función para Gather es:

```
int MPI_Gather(void* mensaje, int contador,
           MPI_Datatype tipo_datos, int raiz,MPI_Comm com);
```

La figura 3.2.2 muestra como se realiza:

Fig. 3.2.2 Gather (recolección)

3.2.6 DISTRIBUCIÓN (SCATTER)

Scatter realiza la operación simétrica a Gather. El proceso raíz posee un vector de elementos, uno para cada proceso del comunicador. Tras realizar la distribución, cada pro-

ceso tiene una copia del vector inicial. Recordemos que MPI permite enviar bloques de datos, no sólo datos individuales. La función es:

```
int MPI_Scatter(void* mensaje, int contador,
          MPI_Datatype tipo_datos, void* destino, int
          raiz,MPI_Comm com);
```

3.2.7. OPERACIONES DE REDUCCIÓN

En el algoritmo Regla del Trapecio, tras la fase de introducción de datos, todos los procesadores ejecutan esencialmente las mismas instrucciones hasta el final de la fase de suma. De este modo, a menos que la función $f(x)$ sea muy complicada (p.ej. que requiera más trabajo ser evaluada en ciertas zonas del intervalo **[a, b]**), esta parte del programa distribuye el trabajo equitativamente entre los procesos. Como hemos dicho antes, éste *no* es el caso en la fase de suma final en donde si utilizamos paso de mensajes el proceso 0 realiza una cantidad de trabajo desproporcionada. Sin embargo probablemente se haya dado cuenta que invirtiendo las flechas de la figura 2.1 podemos usar la comunicación estructurada en árbol. De esta manera podemos distribuir el trabajo de calcular la suma entre los procesos como sigue:

1. (a) 1 envía al 0, 3 al 2, 5 al 4, 7 al 6
   (b) 0 suma su integral a la recibida de 1, 2 suma su integral a la recibida de 3, etc.

2. (a) 2 envía al 0, 6 al 4
   (b) 0 suma, 4 suma

3. (a) 4 envía al 0
   (b) 0 suma

Por supuesto caemos en la misma cuestión que cuando discutíamos la implementación de nuestro propio broadcast: ¿Hace un uso óptimo de la topología de nuestra máquina esta estructura en árbol? Una vez más debemos responder que ello depende de la máquina. Así que, como antes, debemos dejar a MPI hacer dicho trabajo utilizando una función optimizada.

La "suma global" que queremos calcular es un ejemplo de una clase general de operaciones de comunicación colectiva llamadas *operaciones de reducción*. Las operaciones

de reducción forman un subconjunto de las operaciones de comunicación colectiva. En una operación de reducción global todos los procesos (pertenecientes a un comunicador) aportan datos, los cuales se combinan usando una operación binaria. Las operaciones binarias típicas son la suma, el máximo, el mínimo, AND lógico, etc. La función MPI que implementa una operación de reducción es:

```
int MPI_Reduce(void* operando, void* resultado,
           int contador, MPI_Datatype tipo_datos,
           MPI_Op operacion, int raiz, MPI_Comm comunicador);
```

La función MPI_Reduce combina los operandos referenciados por **operando** usando la operación señalada por **operación** y almacena el resultado en la variable **resultado** perteneciente al proceso **raíz**. Tanto **operando** como **resultado** referencian al **contador** posiciones de memoria del tipo **tipo_datos**. **MPI_Reduce()** debe ser llamado por todos los procesos pertenecientes a **comunicador**; además **contador**, **tipo_datos** y **operacion** deben ser iguales para todos los procesos.

El argumento **operación** puede adquirir uno de los valores predefinidos mostrados en el cuadro 3.2.1. También es posible definir operaciones adicionales.

Nombre de operación	**Significado**
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Suma
MPI_PROD	Producto
MPI_LAND	AND Lógico
MPI_BAND	AND Bit a Bit
MPI_LOR	OR Lógico
MPI_BOR	OR Bit a Bit
MPI_LXOR	OR Exclusivo Lógico
MPI_BXOR	OR Exclusivo Bit a Bit
MPI_MAXLOC	Máximo y su Localización
MPI_MINLOC	Minimo y su Localización

Fig. 3.2.1: Operaciones de Reducción

3.2.8 NOTAS IMPORTANTES DE LA IMPLEMENTACIÓN REGLA DEL TRAPECIO

A continuación exponemos la implementación del algoritmo Regla del Trapecio. Notar que cada proceso llama a **MPI_Reduce()** con los mismos argumentos; de hecho, aunque la variable que almacena la suma global sólo tiene significado para el proceso 0, todos los procesos deben suministrar un argumento.

```
void distribuir_datos(int id,long double* pa, long double* pb, long* pn, int*
pnumfuncion){

   1. MPI_Datatype MPI_Tipo_Datos_Entrada;
   2. Tipo_Datos_Entrada datos_entrada;
   3. int raiz=0;
   4. construir_tipo_derivado(&datos_entrada,&MPI_Tipo_Datos_Entrada);
   5. if(id==0){
   6. datos_entrada.a=*pa;
   7. datos_entrada.b=*pb;
   8. datos_entrada.nsegmentos=*pn;
   9. datos_entrada.nfuncion=*pnumfuncion;}
   10. MPI_Bcast(&datos_entrada,1,MPI_Tipo_Datos_Entrada,raiz,MPI_COMM_WORLD);
   11. if(id!=0){
   12. *pa=datos_entrada.a;
   13. *pb=datos_entrada.b;
   14. *pn=datos_entrada.nsegmentos;
   15. *pnumfuncion=datos_entrada.nfuncion;
        }
  }
```

Línea 3: Proceso que envía los datos a los demás procesos

Línea 4: Construimos el tipo MPI Para empaquetar los datos de entrada

Líneas 5-9: Empaquetado de datos; se asignan los datos de entrada a la estructura creada para enviarla al resto de los procesos.

Línea 10: Envía los datos por medio del proceso 0, al resto de los procesos.

Línea 11: Reunión de datos

```
void recolectar_datos(int id, int numprocs, long double* pparcial, long double*
pcompleta){

   1. int raiz=0;
   2. MPI_Reduce(pparcial,pcompleta,1,MPI_LONG_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
}
```

Línea 1: Proceso que recibe las parciales y las sumas

Línea 2: Sumamos las integrales parciales calculadas por cada proceso

3.3 COMUNICADORES Y TOPOLOGIAS

El uso de comunicadores y topologías hace a MPI diferente de la mayoría de los demás sistemas de paso de mensajes. En pocas palabras, un comunicador es una colección de procesos que pueden mandarse mensajes entre ellos. Una topología es una estructura impuesta en los procesos de un comunicador que permite a los procesos ser direccionados de diferentes maneras. Para ilustrar estas ideas desarrollaremos el código que implementa el algoritmo de Fox para multiplicar dos matrices cuadradas.

3.3.1 ALGORITMO MULTIPLICACIÓN DE MATRICES DE FOX

Asumimos que las matrices principales $A = (a_{ij})$ y $B = (b_{ij})$ tienen orden n. También asumimos que el número de procesos p es un cuadrado perfecto, cuya raíz q es tal que el resultado de n/q es un entero. Digamos que $p = q^2$, y que $\bar{n} = n/q$. En el algoritmo de Fox las matrices principales son particionadas en bloques y repartidas entre los procesos. Así vemos nuestros procesos como una tabla virtual de $q \times q$ elementos, donde a cada proceso se le asigna una submatriz $\bar{n} \times \bar{n}$ de cada una de las matrices principales. De manera más formal, tenemos la siguiente correspondencia:

$$\Phi : \{0,1,.....,\, p-1\} \rightarrow \{(s,t) : 0 \le s,t \le q-1\}$$

La siguiente afirmación define nuestra tabla de procesos: el proceso i pertenece a la fila y la columna dadas por ø (i). Además diremos que el proceso con dirección ø$^{-1}$(s,t) tiene asignadas las submatrices:

$$A_{s,t} = \begin{pmatrix} a_{s*\bar{n},t*\bar{n}} & \cdots & a_{(s+1)*\bar{n}-1,t*\bar{n}} \\ \vdots & & \vdots \\ a_{s*\bar{n},(t+1)*\bar{n}-1} & \cdots & a_{(s+1)*\bar{n}-1,(t+1)*\bar{n}-1} \end{pmatrix}$$

$$B_{s,t} = \begin{pmatrix} b_{s*\bar{n},t*\bar{n}} & \cdots & b_{(s+1)*\bar{n}-1,t*\bar{n}} \\ \vdots & & \vdots \\ b_{s*\bar{n},(t+1)*\bar{n}-1} & \cdots & b_{(s+1)*\bar{n}-1,(t+1)*\bar{n}-1} \end{pmatrix}$$

Proceso 0 $A_{00} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix}$	Proceso 1 $A_{01} = \begin{pmatrix} a_{02} & a_{03} \\ a_{12} & a_{13} \end{pmatrix}$	Proceso 2 $A_{02} = \begin{pmatrix} a_{04} & a_{05} \\ a_{14} & a_{15} \end{pmatrix}$
Proceso 3 $A_{10} = \begin{pmatrix} a_{20} & a_{21} \\ a_{30} & a_{31} \end{pmatrix}$	Proceso 4 $A_{11} = \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}$	Proceso 5 $A_{12} = \begin{pmatrix} a_{24} & a_{25} \\ a_{34} & a_{35} \end{pmatrix}$
Proceso 6 $A_{20} = \begin{pmatrix} a_{40} & a_{41} \\ a_{50} & a_{51} \end{pmatrix}$	Proceso 7 $A_{21} = \begin{pmatrix} a_{42} & a_{43} \\ a_{30} & a_{53} \end{pmatrix}$	Proceso 8 $A_{22} = \begin{pmatrix} a_{44} & a_{45} \\ a_{54} & a_{55} \end{pmatrix}$

Cuadro 3.3.1: Partición Matriz 6*6 en 9 procesos

Por ejemplo, si $p = 9$, $\Phi(x) = (x/3, x \bmod 3)$ y $(n = 6)$ entonces la matriz A estaría particionada como se indica en el cuadro 3.3.1.

En el algoritmo de Fox las submatrices A_{rs} y B_{st}, $s = 0,1,....,q-1$ son multiplicadas y acumuladas por el proceso $\Phi^{-1}(r,t)$.

3.3.2 ALGORITMO BÁSICO

Desde $paso = 0$ hasta $paso = q - 1$ (aumentar paso en cada iteración):

1. Elegir una submatriz de A de cada una de las filas de procesos. La submatriz elegida en la fila ***r*** será $A_{r,u}$ donde

$$u = (r + paso) \bmod q$$

2. En cada una de las filas de procesos hacer un broadcast de la submatriz elegida hacia los demás procesos en dicha fila.
3. En cada proceso multiplicar la submatriz recibida de A por la matriz de B que reside en dicho proceso, sin modificar ninguna de estas matrices. Sumar el resultado de la multiplicación al valor que contiene el proceso en la matriz resultado C.
4. En cada proceso enviar la submatriz de B al proceso que está encima de él (los procesos de la primera fila enviarán la submatriz a la última fila).

3.3.3 COMUNICADORES

Si intentamos implementar el algoritmo de Fox, parece claro que nuestro trabajo se facilitaría si pudiéramos tratar ciertos subconjuntos de procesos como el universo de la comunicación, al menos temporalmente. Por ejemplo, en el pseudocódigo.

---

2. En cada una de las filas de procesos hacer un broadcast de la submatriz elegida hacia los demás procesos, en dicha fila.
Sería útil tratar cada fila como el universo de la comunicación, así como la sentencia

---

4. En cada proceso enviar la submatriz B al proceso que está encima de él (los procesos de la primera fila enviarán la submatriz a la ultima fila).
Sería útil tratar cada columna de procesos como el universo de la comunicación.

El mecanismo que MPI proporciona para tratar a un subconjunto de procesos como el universo de la comunicación son los comunicadores. Hasta ahora hemos definido al comunicador como una colección de procesos que pueden mandarse mensajes entre ellos. Ahora que queremos construir nuestros propios comunicadores necesitaremos una discusión más profunda.

En MPI existen dos tipos de comunicadores: *intracomunicadores* e *intercomunicadores*. Los intracomunicadores son esencialmente una colección de procesos que pueden mandarse mensajes entre ellos y utilizar operaciones de comunicación colectivas. Por ejemplo, **MPI_COMM_WORLD** es un intracomunicador, y queremos formar para cada fila y cada columna del algoritmo de Fox otro intracomunicador. Los intercomunicadores, como su propio nombre indica, son utilizados para mandar mensajes entre los procesos

pertenecientes a intracomunicadores *disjuntos*. Por ejemplo, un intercomunicador podría ser útil en un entorno que nos permitiera crear procesos dinámicamente: un nuevo conjunto de procesos que formen un intracomunicador podría ser enlazado al conjunto de procesos original (por ejemplo **MPI_COMM_WORLD**) mediante un intercomunicador. Nosotros sólo utilizaremos intracomunicadores.

Un intracomunicador mínimo está compuesto de:

- Un Grupo, y
- Un Contexto

Un grupo es una colección ordenada de procesos. Si un grupo consiste en ***p*** procesos, a cada proceso en el grupo se le asigna un único identificador, que no es más que un entero no negativo en el rango $0,1,...p-1$. Un contexto puede ser entendido como una etiqueta definida por el sistema que es atribuida al grupo. De este modo dos procesos que pertenecen al mismo grupo y usan el mismo contexto pueden comunicarse. Esta paridad grupo-contexto es la forma más básica de comunicador. Otros datos pueden ser asociados al comunicador. En particular, se le puede asociar una estructura o topología al comunicador, permitiendo un esquema de direccionamiento de los procesos más natural. Abordaremos las topologías mas adelante.

3.3.4 TRABAJANDO CON GRUPOS, CONTEXTOS Y COMUNICADORES

Para ilustrar las bases del uso de los comunicadores, vamos a crear un comunicador cuyo grupo consiste en los procesos de la primera fila de nuestra tabla virtual. Supongamos que **MPI_COMM_WORLD** contiene ***p*** procesos, donde $q^2 = p$**.** Supongamos también que $\Phi(x) = (x/q, x \bmod q)$. Así la primera fila de procesos consistirá en los procesos cuyo identificador es $0,1,...,q-1$ (aquí los identificadores pertenecen a **MPI_COMM_WORLD**). Para crear el grupo de nuestro nuevo comunicador podemos ejecutar el siguiente código:

```
MPI_Group MPI_GROUP_WORLD;
MPI_Group grupo_primera_fila;
MPI_Comm com_primera_fila;
int tam_fila;
int* ids_procesos;
```

```
/* Crea lista con los procesos del nuevo comunicador */
ids_procesos=(int*) malloc(q*sizeof(int));
for (proc=0; proc<q ; proc++)
       ids_procesos[proc]=proc;

/* Obtiene grupos del comunicador mpi_comm_world */
MPI_Comm_group(MPI_COMM_WORLD,&MPI_GROUP_WORLD);

/* Crea el nuevo grupo */
MPI_Group_incl(MPI_GROUP_WORLD,q,ids_procesos,
                    &grupo_primera_fila);

/* Crea el nuevo comunicador */
MPI_Comm_create(MPI_COMM_WORLD,grupo_primera_fila,&com_primera_fila)
```

Este código construye de una manera muy estricta el nuevo comunicador. Primero crea una lista de los procesos que deberán ser asignados al nuevo comunicador. Luego crea un grupo con dichos procesos. Para lo cual requiere dos llamadas; primero se obtiene el grupo asociado a **MPI_COMM_WORLD**, dado que éste es el grupo del cual se extraen los procesos para el nuevo grupo; luego se crea el grupo con **MPI_Group_incl()**. Finalmente el nuevo comunicador es creado con una llamada a **MPI_Comm_create()**. La llamada a **MPI_Comm_create()** asocia implícitamente un contexto al nuevo grupo. El resultado es el comunicador **com_primera_fila**. Ahora los procesos pertenecientes a **com_primera_fila** pueden ejecutar operaciones de comunicación colectivas. Por ejemplo, el proceso 0 (en **grupo_primera_fila**) puede hacer un broadcast de su matriz A_{00} a los otros procesos del grupo **grupo_primera_fila**:

```
int mi_id_en_primera_fila;
float* A_00;

/* mi_id ES EL IDENT. DEL PROCESO EN MPI_GROUP_WORLD */
if(mi_id<q){
MPI_Comm_rank(com_primera_fila,&mi_id_en_primera_fila);

/* ASIGNAMOS ESPACIO A A_00, ORDEN=n */
A_00=(float*) malloc(n*n*sizeof(float));
if(mi_id_en_primera_fila==0){
      /*INICIALIZAMOS A_00 */
```

```
      ...
}
      MPI_Bcast(A_00,n*n,MPI_FLOAT,0,com_primera_fila);
}
```

Los grupos y los comunicadores son *objetos opacos*. Desde un punto de vista práctico significa que los detalles de su representación interna dependen de la implementación MPI particular, y como consecuencia el usuario no puede acceder directamente a ellos. En vez de esto el usuario accede a un *manejador* que referencia al objeto opaco, de manera que dichos objetos opacos son manipulados por funciones MPI especiales, como por ejemplo **MPI_Comm_create()**, **MPI_Group_incl()** y **MPI_Comm_group()**.
Los contextos no son explícitamente usados en ninguna función MPI. En vez de ello, son implícitamente asociados a los grupos cuando los comunicadores son creados. La sintaxis de los comandos que hemos usado para crear **com_primera_fila** es bastante sencilla. El primer comando

```
int MPI_Comm_group(MPI_Comm com, MPI_Group* grupo)
```

simplemente retorna el grupo perteneciente al comunicador **com**.

El segundo comando

```
int MPI_Group_incl(MPI_Group antiguo_grupo,
                   int tamano_nuevo_grupo,
                   int* ids_antiguo_grupo,
                   MPI_Group* nuevo_grupo)
```

crea un nuevo grupo a partir de una lista de procesos pertenecientes al grupo existente **antiguo_grupo**. El número de procesos en el nuevo grupo es **tamano_nuevo_grupo**, y los procesos que serán incluidos son listados en **ids_antiguo_grupo**. El proceso 0 en **nuevo_grupo** tiene el identificador **ids_antiguo_grupo[**0**]** en **antiguo_grupo**, el proceso 1 en **nuevo_grupo** es el **ids_antiguo_grupo[**1**]** en **antiguo_grupo**, y así sucesivamente.
El comando final

```
int MPI_Comm_create(MPI_Comm antiguo_com,
                    MPI_Group nuevo_grupo,
```

```
MPI_Comm* nuevo_com)
```

asocia el contexto al grupo **nuevo_grupo**, y crea el comunicador **nuevo_com**. Todos los procesos en **nuevo_grupo** pertenecen al grupo subyacente **antiguo_com**.
Existe una diferencia extremadamente importante entre las primeras dos funciones y la tercera. **MPI_Comm_group()** y **MPI_Group_incl()** son ambas operaciones locales. Ello quiere decir que no hay comunicación entre los procesos implicados en dicha ejecución. Sin embargo **MPI_Comm_create()** *es* una operación colectiva. *Todos* los procesos en **antiguo_grupo** deben llamar a **MPI_Comm_create()** con los mismos argumentos. El estándar MPI da tres razones para ello:

- Permite a la implementación colocar a **MPI_Comm_create()** como la primera de las comunicaciones colectivas regulares.
- Proporciona mayor seguridad.
- Permite que las implementaciones eviten comunicaciones relativas a la creación del contexto.

Notemos que como **MPI_Comm_create()** es colectiva tendrá un comportamiento sincronizador. En particular, si varios comunicadores están siendo creados, todos deben ser creados en el mismo orden en todos los procesos.

3.3.5 PARTICIONAMIENTO DE LOS COMUNICADORES

En nuestro programa de multiplicación de matrices necesitamos crear múltiples comunicadores (uno para cada fila de procesos y uno para cada columna). Éste podría ser un proceso extremadamente tedioso si *p* fuera grande y tuviéramos que crear cada comunicador usando las tres funciones comentadas en la sección anterior. Afortunadamente MPI proporciona una función, **MPI_Comm_split()**, que puede crear varios comunicadores simultáneamente.
Para ejemplificar su uso crearemos un comunicador para cada fila de procesos:

```
MPI_Comm com_fila;
int fila;
/* id ES EL IDENTIFICADOR EN MPI_COMM_WORLD.
 * q*q = p */
fila = id / q;
```

```
MPI_Comm_split(MPI_COMM_WORLD, fila, id, &com_fila);
```

Una única llamada a **MPI_Comm_split()** crea *q* nuevos comunicadores, todos ellos con el mismo nombre **com_fila**. Por ejemplo, si $p = 9$ el grupo **com_fila** consistirá en los procesos 0, 1 y 2 para los procesos 0, 1 y 2. En los procesos 3, 4 y 5 el grupo subyacente a **nuevo_com** será el formado por los procesos 3, 4 y 5; y lo mismo ocurrirá con los procesos 6, 7 y 8.

La sintaxis de **MPI_Comm_split()** es:

```
int MPI_Comm_split(MPI_Comm antiguo_com,
                   int clave_particion,
                   int clave_id, MPI_Comm* nuevo_com)
```

La llamada crea un comunicador para cada valor de **clave_particion**. Los procesos con el mismo valor en **clave_particion** forman un nuevo grupo. El identificador de los procesos en el nuevo grupo estará determinado por su valor en **clave_id**. Si los procesos A y B llaman ambos a la función **MPI_Comm_split()** con el mismo valor en **clave_particion**, y el argumento **clave_id** pasado por el proceso A es menor que el pasado por el proceso B, entonces el identificador de A en el grupo **nuevo_com** será menor que el identificador del proceso B. Si llaman a la función con el mismo valor en **clave_id** el sistema asignará arbitrariamente a uno de los procesos un identificador menor.
MPI_Comm_split() es una operación colectiva y debe ser llamada por todos los procesos pertenecientes a **antiguo_com**. La función puede ser utilizada incluso si no deseamos asignarle un comunicador a todos los procesos. Lo que puede ser realizado pasando la constante predefinida **MPI_UNDEFINED** en el argumento **clave_particion**. Los procesos que hagan esto obtendrán como valor de retorno en **nuevo_com** el valor **MPI_COMM_NULL**.

3.3.6 TOPOLOGIAS

Recordemos que es posible asociar información adicional (más allá del grupo y del contexto) a un comunicador. Una de las piezas de información más importantes que se pueden adjuntar al comunicador es la topología. En MPI una *topología* no es más que un mecanismo para asociar diferentes esquemas de direccionamiento a los procesos perte-

necientes a un grupo. Notar que las topologías en MPI son topologías *virtuales*, lo que quiere decir que podría no haber una relación simple entre la estructura de procesos de una topología virtual y la estructura física real de la máquina paralela.

Esencialmente existen dos tipos de topologías virtuales que se pueden crear en MPI: la topología *cartesiana* o rejilla, y la topología *gráfica*. Conceptualmente ésta última engloba a la primera. De todos modos, y debido a la importancia de las rejillas en las aplicaciones, existe una colección separada de funciones en MPI cuyo propósito es la manipulación de rejillas virtuales.

En el algoritmo de Fox queremos identificar los procesos en **MPI_COMM_WORLD** mediante las coordenadas de una rejilla, y cada fila y cada columna de la rejilla necesitan formar su propio comunicador. Observemos un método para construir dicha estructura.

Comenzamos asociando una estructura de rejilla a **MPI_COMM_WORLD**. Para hacer esto necesitamos especificar la siguiente información:

1. El número de dimensiones de la rejilla. Tenemos 2.
2. El tamaño de cada dimensión. En nuestro caso, no es más que el número de filas y el número de columnas. Tenemos q filas y q columnas.
3. La periodicidad de cada dimensión. En nuestro caso esta información especifica si la primera entrada de cada fila o columna es "adyacente" a la última entrada de dicha fila o columna, respectivamente. Dado que nosotros queremos un paso "circular" de las matrices en cada columna, sería provechoso que la segunda dimensión fuera periódica. No es importante si la primera dimensión es periódica o no.
4. Finalmente MPI ofrece al usuario la posibilidad de permitir al sistema optimizar el direccionamiento de la rejilla, posiblemente reordenando los procesos pertenecientes al comunicador asociado, para así aprovechar mejor la estructura física de dichos procesos. Dado que no necesitamos preservar el orden de los procesos en **MPI_COMM_WORLD**, deberíamos permitir al sistema el reordenamiento.

Para implementar estas decisiones simplemente ejecutaremos el siguiente código:

```
MPI_Comm com_rejilla;

int dimensiones[2];
int periodicidad[2];
```

```
int reordenamiento = 1;

dimensiones[0] = dimensiones[1] = q;
periodicidad[0] = periodicidad[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dimensiones,
                periodicidad, reordenamiento,&com_rejilla);
```

Tras ejecutar este código el comunicador **com_rejilla** contendrá los procesos pertenecientes a **MPI_COMM_WORLD** (posiblemente reordenados) y tendrá asociado un sistema de coordenadas cartesianas de dos dimensiones. Para determinar las coordenadas de un proceso simplemente llamaremos a la función **MPI_Cart_coords()**:

```
int coordenadas[2];
int id_rejilla;

MPI_Comm_rank(com_rejilla, &id_rejilla);
MPI_Cart_coords(com_rejilla, id_rejilla, 2, coordenadas);
```

Observemos que necesitamos llamar a **MPI_Comm_rank()** para obtener el identificador del proceso en **com_rejilla**. Esto es necesario porque en nuestra llamada a **MPI_Cart_create()** establecimos la variable reordenamiento a 1, y por lo tanto el identificador original de un proceso en **MPI_COMM_WORLD** puede cambiar en **com_rejilla**.

La "inversa" a **MPI_Cart_coords()** es **MPI_Cart_rank()**:

```
int MPI_Cart_rank(com_rejilla, coordenadas, &id_rejilla)
```

Dadas las coordenadas de un proceso, **MPI_Cart_rank()** retorna el identificador del proceso en su tercer parámetro **id_proceso**.

La sintaxis de **MPI_Cart_create()** es:

```
int MPI_Cart_create(MPI_Comm antiguo_com,
                    int numero_dimensiones,
                    int* tam_dimensiones,
```

```
                    int* periodicidades,
                    int reordenamiento,
                    MPI_Comm* com_cartesiano)
```

MPI_Cart_create() crea un nuevo comunicador, **com_cartesiano**, aplicando una topología cartesiana a **antiguo_com**. La información sobre la estructura de la topología cartesiana está contenida en los parámetros **numero_dimensiones**, **tam_dimensiones** y **periodicidades**. El primero de ellos, **numero_dimensiones**, contiene el número de dimensiones que forman el sistema de coordenadas cartesianas. Los dos siguientes, **tam_dimensiones** y **periodicidades**, son:

3	4	5
0	1	2
6	7	8

Cuadro 3.3.2 Topología Física 3*3

vectores que tienen el mismo orden que **numero_dimensiones**. El vector **tam_dimensiones** especifica el orden de cada dimensión, mientras que el vector **periodicidades** especifica para cada dimensión si es circular o linear.

Los procesos pertenecientes a **com_cartesiano** son ordenados de fila en fila. De este modo la primera fila contiene los procesos $0,1,....,tam\_\dim ensiones[0]-1$, la segunda fila contendrá:

$tam\_\dim ensiones[0]$, $tam\_\dim ensiones[0]+1,....,2*tam\_\dim ensiones[0]-1$ y así sucesivamente. En este sentido podría ser ventajoso cambiar el orden relativo de los procesos en **com_antiguo**. Por ejemplo, supongamos que la topología *física* es una tabla 3*3 y que los procesos pertenecientes **com_antiguo**, representados por sus identificadores, están asociados a los procesadores como se expone en el cuadro 3.3.2

Claramente la eficiencia del algoritmo de Fox se mejoraría si reordenáramos los procesos. Sin embargo, y dado que el usuario no conoce cómo se asocian los procesos a los procesadores, deberíamos dejar al sistema hacerlo estableciendo el parámetro **reordenamiento** a 1.

Dado que **MPI_Cart_create()** construye un nuevo comunicador, se trata de una operación colectiva.

La sintaxis de las funciones que nos informan sobre el direccionamiento de los procesos es:

```
int MPI_Cart_rank(MPI_Comm cart_com, int* coordenadas, int* id)
```

```
int MPI_Cart_coords(MPI_Comm cart_com, int id,
                    int numero_dimensiones, int* coordenadas)
```

La función **MPI_Cart_rank()** retorna el identificador en **cart_com** del proceso que tiene las coordenadas cartesianas representadas en el vector **coordenadas**. Así, **coordenadas** es un vector de orden igual al número de dimensiones de la topología cartesiana asociada a **cart_com**. **MPI_Cart_coords()** es la inversa a **MPI_Cart_rank()**: retorna las coordenadas del proceso que tiene como identificador **id** en el comunicador cartesiano **cart_com**. Notar que ambas funciones son locales.

3.3.7 DIVISION DE REJILLAS

También se puede dividir una rejilla en rejillas de menores dimensiones. Por ejemplo, podemos crear un comunicador para cada fila de la rejilla como sigue:

```
int var_coords[2];

MPI_Comm com_fila;

var_coords[0]=0; var_coords[1]=1;
MPI_Cart_sub(cart_com,var_coords,&com_fila);
```

La llamada a **MPI_Cart_sub()** crea q nuevos comunicadores. El argumento **var_coords** es un vector de booleanos. Especifica para cada dimensión si "pertenece" al nuevo comunicador. Dado que estamos creando comunicadores para las filas de la rejilla, cada nuevo comunicador consiste en los procesos obtenidos al fijar la coordenada de la fila, permitiendo que la coordenada de la columna varíe. Con este propósito le asignamos a **var_coords[**0**]** el valor 0 (la primera coordenada no varía) y le asignamos a **var_coords[**1**]** el valor 1 (la segunda coordenada varía). En cada proceso retornamos el nuevo comunicador en **com_fila**. Si queremos crear comunicadores para las columnas, simplemente invertimos los valores en los elementos de **var_coords**.

```
MPI_Com com_columna;

var_coords[0]=1; var_coords[1]=0;
MPI_Cart_sub(cart_com,var_coords,&com_columna);
```

Notar la similaridad entre **MPI_Cart_sub()** y **MPI_Comm_split()**. Ambas llevan a cabo funciones similares (las dos dividen un comunicador en una colección de nuevos comunicadores). Sin embargo **MPI_Cart_sub()** sólo puede ser usada en comunicadores que tengan asociada una topología cartesiana, y el nuevo comunicador sólo puede ser creado fijando (o variando) una o más dimensiones del antiguo comunicador. Notar también que **MPI_Cart_sub()** es, como **MPI_Comm_split()**, una operación colectiva.

3.3.8 IMPLEMENTACION MULTIPLICACION DE MATRICES DE FOX

En esta sección mostramos la implementación del algoritmo de Fox. En este programa la función **Config_cart()** genera varios comunicadores e información asociada a ellos. Debido a que este procedimiento requiere muchas variables, y dado que la información contenida en dichas variables es necesaria en la ejecución de otras funciones, utilizaremos una estructura tipo **Info_Cart** para agrupar toda esa información generada. Luego realizaremos la multiplicación de matrices por el método de Fox mediante la llamada a la función **Fox()**; dicha función se apoya en otra, **Mult_Matrices()**, creada para ejecutar la indispensable multiplicación de matrices locales.

3.3.9 NOTAS IMPORTANTES DE LAS MATRICES DE FOX

```
void Fox(Info_cart* pcart,MatrizLocal matriz1, MatrizLocal matriz2,MatrizLocal result,int orden_local){
…..
    1.  temp1=(MatrizLocal*)malloc(sizeof(MatrizLocal));
    2.  origen=(pcart->fila + 1) % pcart->orden_cart;
    3.  destino=(pcart->fila + pcart->orden_cart - 1) % pcart->orden_cart;
    4.  /*MIENTRAS QUEDE PROCESOS EN LA FILA...*/
    5.  for(i=0;i<(pcart->orden_cart);i++){
    6.  bcast_raiz=(pcart->fila + i)%(pcart->orden_cart);
    7.  if(bcast_raiz==pcart->columna){
    8.  MPI_Bcast(matriz1,1,Tipo_envio_matriz,bcast_raiz,pcart->comm_fila);
    9.  mult_matrices(matriz1,matriz2,result,orden_local);
    10. }
    11. else{
    12. MPI_Bcast(temp1,1,Tipo_envio_matriz,bcast_raiz,pcart->comm_fila);
    13. mult_matrices(*temp1,matriz2,result,orden_local);
    14. }
    15. MPI_Sendrecv_replace(matriz2,1,Tipo_envio_matriz,destino,etiqueta,origen,etiqueta,pcart-
        >comm_columna,&status);
  }
}
```

Líneas 1: Reservamos espacio para las matrices.
Líneas 2 y 3: Establecemos el numero de fila de los procesos origen y destino d la matriz local 2.

Líneas 7 - 9: Si es el turno del propio proceso en ejecución, se envía matriz_local 1 hacia los procesos de la misma fila y se multiplica localmente.
Línea 15: Por ultimo se envía hacia arriba en la columna la matriz local 2.

CAPITULO 4
ANALISIS DE LOS RESULTADOS

INTRODUCCIÓN

En el presente capítulo, se evaluarán los algoritmos diseñados en esta tesis, para ello, se usan los siguientes criterios; el tiempo de ejecución, y el número de procesadores usados. Cabe mencionar que aparte de estas medidas estándar pueden utilizarse otras muchas medidas relacionadas con el hardware, aunque sólo deben ser empleadas cuando se conoce bien el entorno particular en donde dicho algoritmo es ejecutado. A continuación describimos cada una de ellas.

4.1 TIEMPO DE PROCESAMIENTO

Definimos al **tiempo de procesamiento** como el tiempo que emplea la computadora en calcular el resultado, eliminando el tiempo de inicialización y finalización de los procesos, la comunicación con el usuario. De esta manera si el usuario tarda diez segundos en introducir los datos para un problema, ese tiempo no será añadido al tiempo de procesamiento pero sí al tiempo de ejecución. La función **MPI_Wtime( )** nos ayudará a calcular el tiempo de procesamiento de un programa para después mostrarlo al usuario por la salida estándar.

4.1.1 NÚMERO DE PROCESADORES

El siguiente criterio más importante en la evaluación de un algoritmo es el *número de procesadores* que requiere para resolver un problema. Cuesta dinero comprar, mantener y hacer que funcionen las computadoras. Cuando varios procesadores están presentes el problema del mantenimiento se complica, y el precio necesario para garantizar un alto grado de fiabilidad aumenta de manera considerable. De esta manera, cuanto mayor sea el número de procesadores que utiliza un algoritmo para resolver un problema, más cara será la solución que obtendremos.

4.2. RESULTADOS OBTENIDOS

4.2.1 ALGORITMO CÁLCULO DE AREAS MEDIANTE MONTECARLO

A continuación se presentan una serie de pruebas que se realizaron en el cluster, utilizando el mismo número de muestras pero cambiando el número de procesadores, además se presenta el tiempo de procesamiento, en la Fig. 4.1.

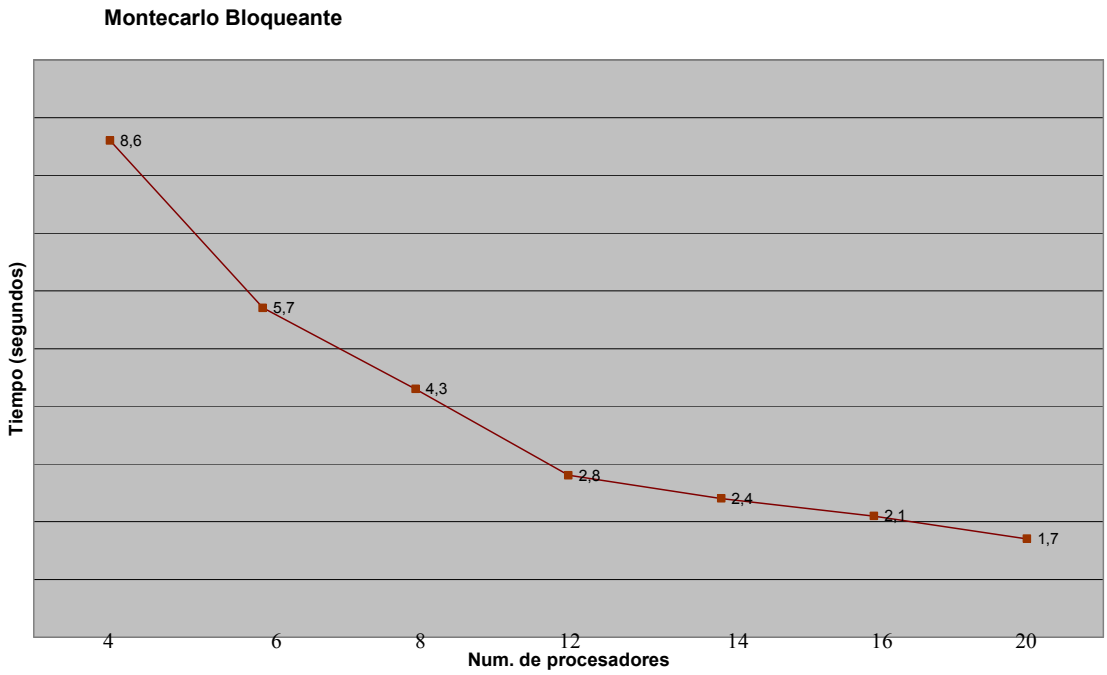

Fig. 4.1 Montecarlo Bloqueante

Los datos graficados se presentan en la siguiente tabla:

Número de procesadores utilizados	Número de muestras	Tiempo de procesamiento
4	200000000 misma para todos los casos.	8.600231 seg.
6		5.720601 seg.
8		4.309154 seg.
12		2.877757 seg.
14		2.402061 seg.
16		2.10254 seg.
20		1.700652 seg.

Tabla 4.1 Datos, Montecarlo Bloqueante

Ahora solo variamos el número de muestras y dejamos el número de procesadores fijo.

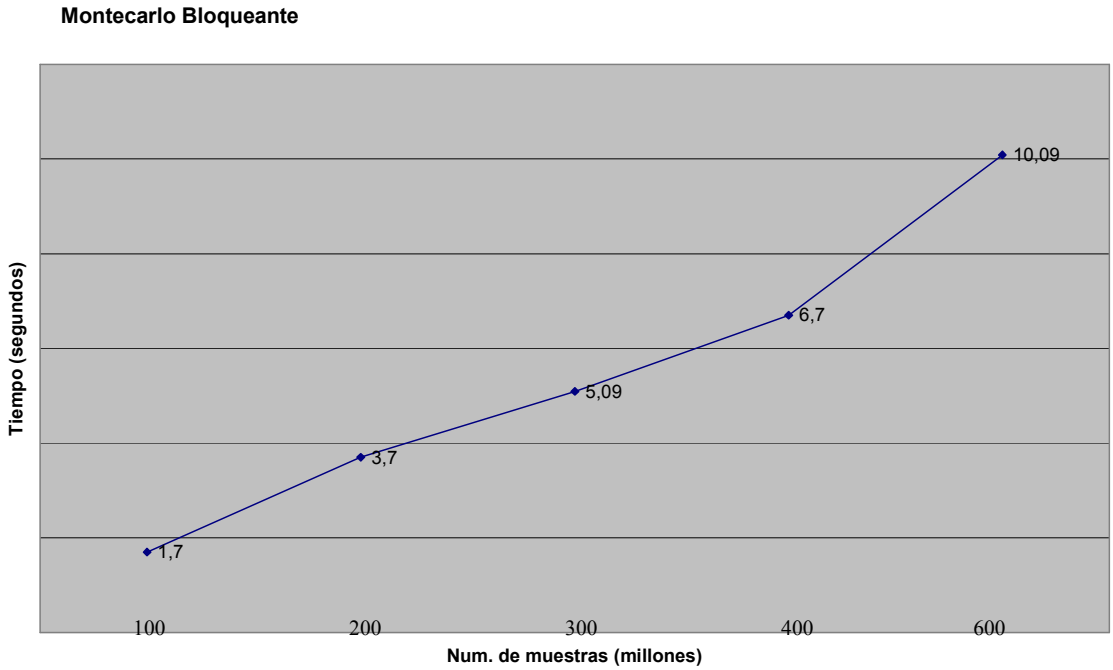

Fig. 4.1.1. Montecarlo Bloqueante, mismo número de procesadores

Los datos se presentan en la tabla siguiente:

Número de procesadores utilizados	Número de muestras (millones)	Tiempo de procesamiento(seg)
10 procesadores para todos los casos.	100	1.767568 seg.
	200	3.706939 seg.
	300	5. 097444 seg.
	400	6.714262 seg.
	600	10.094262 seg.

Tabla 4.1.1. Montecarlo Bloqueante, mismo número de procesadores

4.2.2 ANALISIS DE LOS RESULTADOS

El modelo de ejecución de este algoritmo se interpreta de la siguiente manera: Iniciamos una vez que el proceso 0 distribuye los datos de entrada a los demás procesos enviando a cada uno un mensaje bloqueante. Una vez realizado este paso se produce el procesamiento interno en cada uno de los procesos durante el cual no existe comunicación entre los procesos. Cuando los procesos culminan el tiempo de procesamiento envían su resultado local al proceso 0 mediante un mensaje bloqueante. El proceso 0 recoge dichos resultados locales, calcula la media de todos ellos e imprime el resultado global. Una vez hecho esto el programa termina. Los resultados de la fig. 4.1 muestran que cuantos más procesadores intervienen en la ejecución es el tiempo de ejecución mejora conside-

rablemente. De lo cual podemos concluir que deberíamos emplear más o menos procesadores dependiendo de las necesidades de velocidad de procesamiento que se requiera.

Por otro lado, de los resultados de la fig. 4.1.1 es digno de observar que cuando conservamos el mismo número de procesadores y variamos el número de muestras, el tiempo de procesamiento aumenta, lo cual es razonable pues el trabajo aumenta pero no así el número de trabajadores lo que nos muestra que se debe buscar un equilibrio entre el número de muestras y el número de procesadores según las necesidades de procesamiento que se requiera en el manejo de datos, buscando el mejor desempeño el menor tiempo de procesamiento para nuestras aplicaciones.

4.2.3 ALGORITMO CÁLCULO DE AREAS MEDIANTE MONTECARLO NO BLOQUEANTE

A continuación se presentan una serie de pruebas que se realizaron en el cluster, variando el número de muestras y conservando el número de procesadores, se muestra el tiempo de procesamiento, pero ahora con las funciones no bloqueantes.

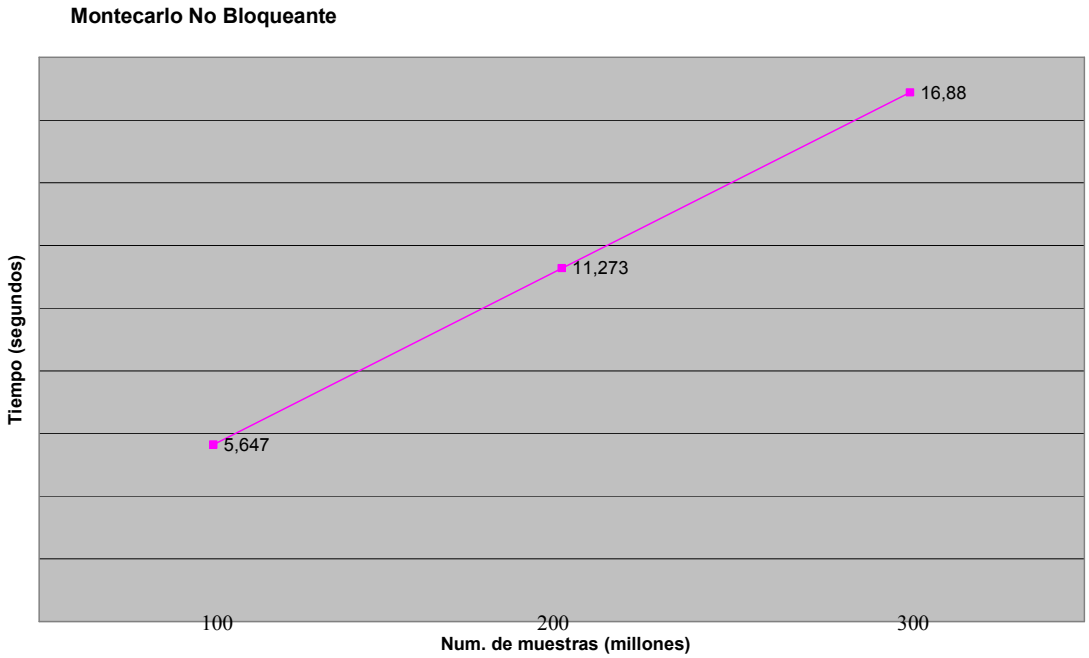

Fig. 4.2.1 Montecarlo No bloqueante

Los datos graficados se muestran en la tabla siguiente:

Número de muestras (millones)	Número de procesadores	Tiempo de procesamiento
100	3 procesadores mismos para todos los casos.	5.647816 seg.
200		11.273314 seg.
300		16.881390 seg.

Tabla 4.2.1 Montecarlo No Bloqueando, mismo número de procesadores

Ahora conservamos el mismo número de muestras, a saber, 100000000, y variamos el número de procesadores, a continuación se muestra el gráfico resultante.

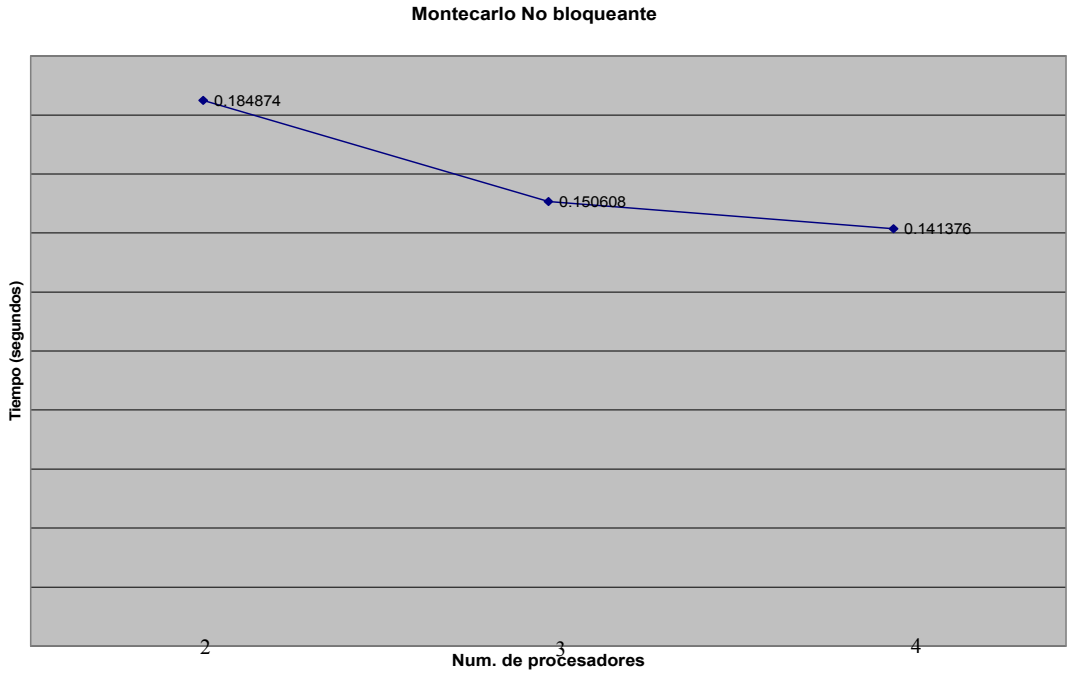

Fig.4.2.1.1 Montecarlo No bloqueante, mismo número de muestras

Los datos graficados se muestran en la siguiente tabla:

Número de muestras (millones)	Número de procesadores	Tiempo de procesamiento
100	2	0.184874 seg.
	3	0.156080 seg.
	4	0.141376 seg.

Tabla 4.2.1.1 Montecarlo No Bloqueante, mismo número de muestras

4.2.4 ANALISIS DE LOS RESULTADOS

El modelo de ejecución de este algoritmo se interpreta de manera similar al algoritmo bloqueante, con la diferencia de que el paso de mensaje se lleva a cabo de forma no bloqueante, dado los resultados obtenidos, podemos ver que conforme aumenta el número de muestras, aumentan respectivamente el tiempo de procesamiento, lo cual es correcto dado que el trabajo aumenta, sin embargo el número de trabajadores es el mismo.

Por otro lado, el gráfico 4.2.1.1 muestra que al conservar el número de muestras y aumentar el número de procesadores, el tiempo de procesamiento disminuye de manera considerable por lo tanto podemos concluir que para obtener resultados óptimos es necesario equilibrar el número de muestras con el número de procesadores según, las necesidades de procesamiento que se requiera.

4.3 ALGORITMO REGLA DEL TRAPECIO

El modelo de ejecución de este algoritmo se interpreta de la siguiente manera. En un principio se produce una operación *broadcast*, con la cual distribuimos los datos de entrada desde el proceso 0 hacia los demás procesos. Una vez realizado este paso se produce el procesamiento interno en cada uno de los procesos durante el cual no existe comunicación entre los procesos. Cuando cada proceso ha culminado su propio procesamiento el proceso 0 recoge los resultados de cada proceso a través de una operación *reduce*. Luego el proceso 0 imprime los resultados y el programa termina.

A continuación se presentan un gráfico de la serie de resultados de las pruebas realizadas en el cluster, utilizando el mismo número de segmentos y variando el número de procesadores.

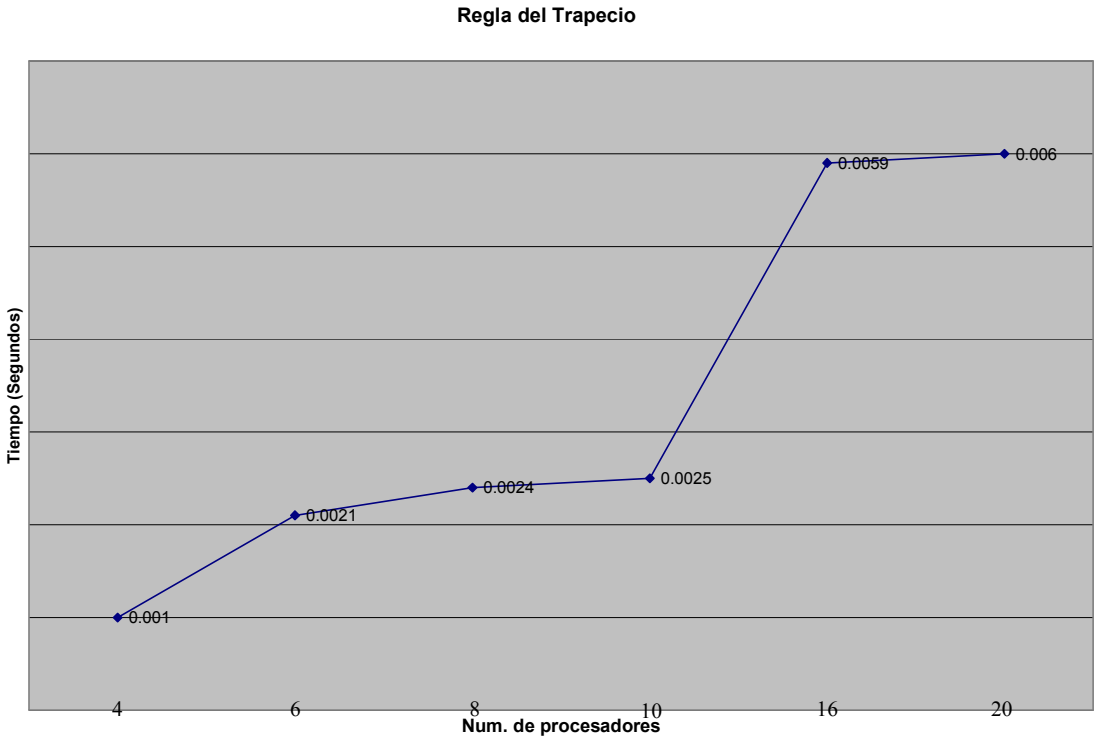

Fig.4.3 Regla del Trapecio

A continuación se presentan los datos graficados en la tabla siguiente:

Número de procesadores utilizados	Número de muestras(millones)	Tiempo de procesamiento
4	2800 misma para todos los casos.	0.0010 seg.
6		0.0021seg.
8		0.0024 seg.
10		0.0025 seg.
16		0.0059 seg.
20		0.0060 seg.

Tabla 4.3 Datos Regla del Trapecio

4.3.1 ANÁLISIS DE LOS RESULTADOS

Dado los resultados obtenidos se puede ver que, cuantos más procesadores intervienen en la ejecución mayor es el tiempo de ejecución. De lo cual podemos deducir que no es recomendable utilizar el cluster para ejecutar aplicaciones pequeñas, pues los recursos se desperdician de manera considerable, más bien utilizar esta tecnología en el manejo de grandes cantidades de datos.

4.4 ALGORITMO MATRICES DE FOX

El algoritmo que presentamos ahora tiene una serie de características que lo diferencian del resto. Para evaluarlo debemos tener en cuenta una serie de factores. El primer factor es la sobrecarga en la comunicación. Para multiplicar matrices de orden N utilizando P procesos, este algoritmo necesita hacer P operaciones de broadcast $P * \sqrt{P}$ y operaciones simples de paso de mensajes. El tamaño de cada uno de estos mensajes tampoco es una cuestión a desconsiderar; cada uno de los mensajes deberá pasar una matriz de orden $\frac{N}{\sqrt{P}}$, dato que coincide con el orden de las matrices locales de cada uno de los procesos.

El segundo factor es que no podemos ejecutarlo con cualquier número de procesadores. El número de procesadores utilizado debe tener raíz entera para que una matriz cuadrada pueda ser distribuida equitativamente entre los procesos.

4.4.1 PRUEBAS REALIZADAS EN EL CLUSTER

En la tabla siguiente se muestran los tiempos obtenidos en la ejecución del algoritmo Multiplicación de matrices de Fox, variamos el número de procesadores y conservamos el tamaño de la matriz.

El gráfico resultante es el siguiente:

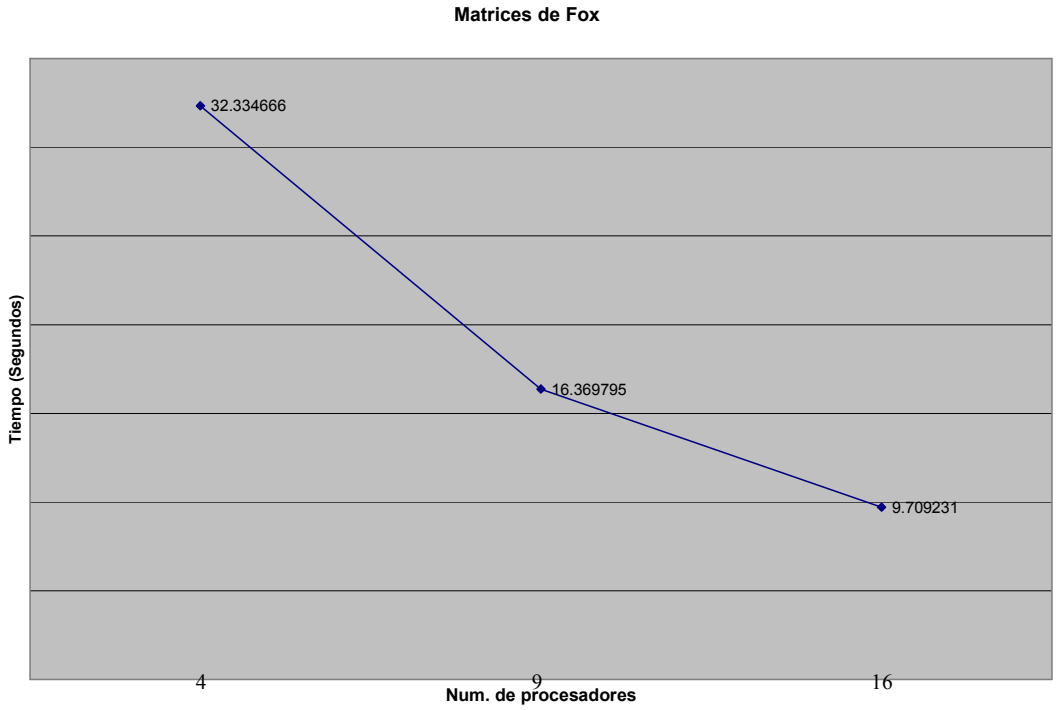

Fig. 4.4 Matrices de Fox

Los datos graficados se muestran en la siguiente tabla:

Número de procesadores utilizados	Orden de la matriz	Tiempo de procesamiento
4	1400 mismo orden para todas las pruebas.	32.334666 seg.
9		16.369795 seg.
16		9.709231 seg.

Tabla4.4 Datos Matrices de Fox

Ahora mismo número de procesadores, variamos el orden de la matriz:

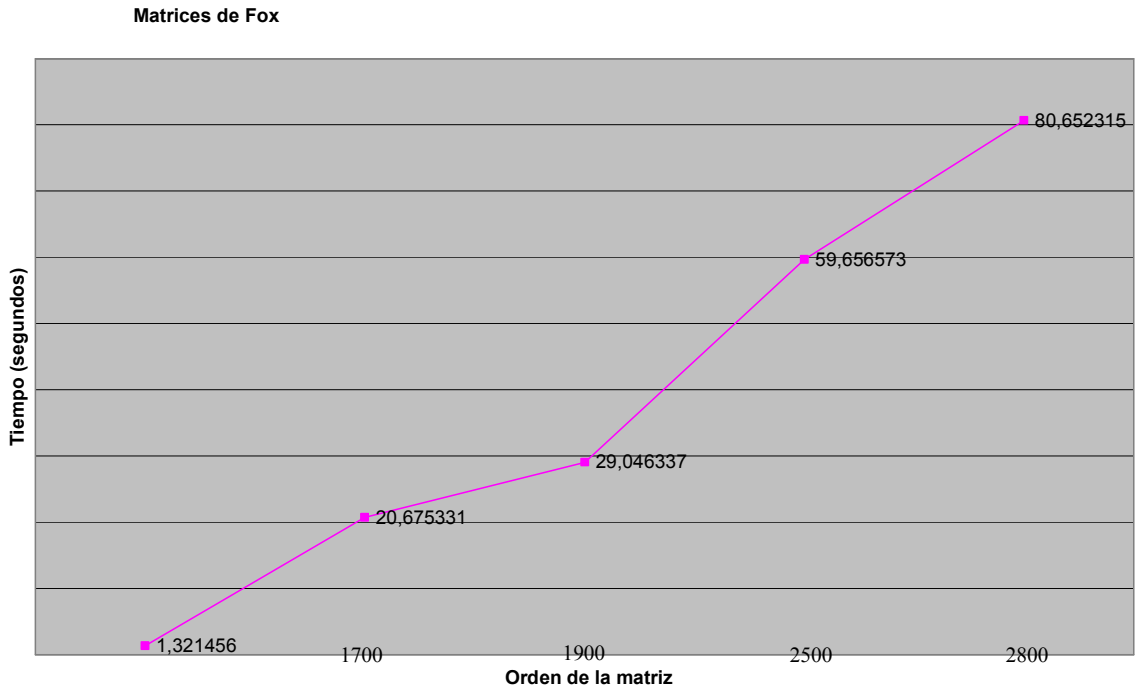

Fig. 4.4.1 Matrices de Fox, mismo número de nodos

Los datos graficados se muestran en la tabla siguiente:

Número de procesadores utilizados	Orden de la matriz	Tiempo de procesamiento
16	1400	9.915678 seg.
	1700	17.376449 seg.
	1900	25.747455 seg.
	2500	56.357691 seg.
	2800	77.353433 seg.

Tabla 4.4.1 Matrices de Fox, mismo número de nodos

4.4.2 ANÁLISIS DE LOS RESULTADOS

El modelo de ejecución de este algoritmo con 16 procesos demuestra la sobrecarga en la comunicación que genera. Cada uno de los mensajes transmitidos tendrá al menos el mismo tamaño que las matrices locales de los procesos. Una vez que los procesos culminan el procesamiento de sus matrices locales, envían los resultados al proceso 0 para generar la salida.

Como podemos observar en la fig. 4.4 la disminución del tiempo de ejecución es evidente, para todas las cantidades de procesos utilizadas; sin embargo dicha disminución es más paulatina cuanto mayor es la cantidad de procesos empleados. Con este planteamiento podemos prever que para tamaños de matrices superiores a los empleados en las pruebas la utilización de 16 procesadores será la más rentable.

En el gráfico 4.4.1 observamos que aunque el número de procesadores se conserva; el tiempo de procesamiento crece conforme crece el orden de la matriz utilizada. De lo cual podemos concluir que para obtener resultados óptimos al procesar datos, debemos encontrar un equilibrio entre el número de procesadores y el tamaño del orden de la matriz, de acuerdo a las necesidades de procesamiento de datos que se requieran.

CONCLUSIONES GENERALES

En esta tesis se desarrolló una investigación sobre el lenguaje de programación MPI, y sus principales características, el hardware que se utilizó para llevar a cabo las pruebas correspondientes fueron realizadas en un cluster que cuenta con las siguientes características: 16 nodos con interconexión a 1 Gbps, donde cada nodo tiene:

- 2 CPU's AMD de la familia Opteron 64 a 2.0 GHz
- 2 GB RAM
- 36 GB HD SCSI

Con la serie de pruebas que se realizaron se puede ver que a mayor número de procesadores empleados para resolver un problema, menor es el tiempo de procesamiento , lo cual demuestra la utilidad y potencialidad del procesamiento paralelo, utilizando un cluster, además se demostró que MPI es un software eficiente para resolver problemas paralelos, al estudiar las características de MPI tales como el paso de mensajes bloqueantes y no bloqueantes en el algoritmo de Montecarlo, el gráfico 4.1.1 muestra que el tiempo mejora en el caso de la versión no bloqueante, con lo que se muestra la eficiencia del paso de mensajes no bloqueantes sobre el paso de mensajes bloqueantes. Una característica más es la capacidad de MPI para realizar broadcast como se pudo notar en el algoritmo de La Regla del Trapecio. Finalmente, en el algoritmo de matrices de Fox, se ejemplificó el uso de topologías además de la flexibilidad de MPI para realizar la creación de comunicadotes por parte del usuario, dado que se implementaron los comunicadores necesarios para el cálculo de matrices, demostrando así, la potencialidad de MPI para el manejo de problemas paralelos.

De lo cual podemos concluir por los resultados obtenidos, que dependiendo de nuestras necesidades de procesamiento de datos utilizaremos el número de nodos correspondiente, pues cuando se realizan operaciones con cantidades pequeñas de datos; como sucedió en el caso del algoritmo de la regla del trapecio, el tiempo de procesamiento no mejoró; por lo que en este tipo de casos no es necesario emplear procesamiento paralelo pues así evitaremos el desperdicio de recursos de nuestra máquina; por lo tanto lo más apropiado es adaptar el número de procesadores a la cantidad de datos que se esté manejando para obtener el tiempo óptimo de

procesamiento, como se pudo ver en los resultados de los gráficos obtenidos de los algoritmos Montecarlo y matrices de Fox, así que en esta tesis se demostró:

- La potencialidad y utilidad del procesamiento paralelo, en el manejo de grandes cantidades de datos.
- Que MPI es un software eficiente para resolver problemas paralelos, tal como se demostró en cada uno de los resultados de los algoritmos graficados.

Por lo tanto se concluye que se cumplió de manera satisfactoria con los objetivos planteados para la realización de ésta tesis.

TRABAJO A FUTURO

Dado que esta investigación abarcó las características más importantes de MPI, se desprende que puede ser utilizado en un futuro como:

- Base para futuras investigaciones, facilitando el uso de la información y el software.
- Implementar una interfaz.
- Implementación de la herramienta XMPI, para el análisis y la depuración de las aplicaciones.
- Implementar los programas de forma gráfica.

- **BIBLIOGRAFIA**

[1] High Performance
Cluster Computing
Architectures and systems, Volume 1
Rajkumar Buyya
ED. Prentice Hall PTR.

[2] Los CLUSTERS como plataforma de procesamiento paralelo.
Óscar Pino Morillas.
Roberto Francisco Arroyo Moreno.
Francisco Javier Nievas Muñoz.

[3] Parallel programming in C with MPI and OpenMP
Michael J. Quinn.
McGraw-Hill Higher Education, c2004.

[4] Parallel programming with MPI
Peter S. Pacheco.
Morgan Kaufmann Publishers, c1997.

[5] Parallel scientific computing in C++ and MPI: a seamless approach to parallel algorithms and their implementation
George Em Karniadakis and Robert M. Kirby II.
Cambridge University Press, 2003.

[6] MPI
http://www.acm.org/crossroads/espanol/xrds8-3/programming.html

[7] Arquitectura de los clusters
http://www.josecc.net/archivos/tesis/tesis_html1/node6.html

[8] LAM/MPI Parallel Computing
http://www.lam-mpi.org

[9] MPI Forum
http://www.mpi-forum.org/